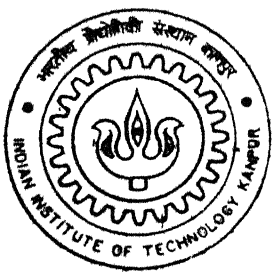
PARALLELISATION OF A 3-D FINITE VOLUME INCOMPRESSIBLE NAVIER STOKES SOLVER FOR COMPLEX GEOMETRY

By

Abir Banerjee



DEPARTMENT OF MECHANICAL ENGINEERING

Indian Institute of Technology Kanpur

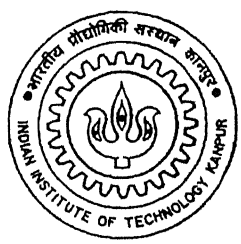
FEBRUARY, 2002

PARALLELISATION OF A 3-D FINITE VOLUME INCOMPRESSIBLE NAVIER-STOKES SOLVER FOR COMPLEX GEOMETRY

A THESIS SUBMITTED
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF TECHNOLOGY

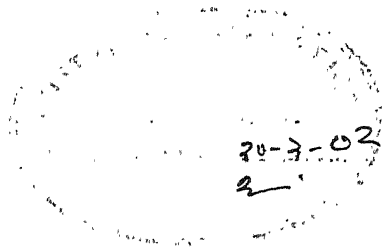
by

ABIR BANERJEE



DEPARTMENT OF MECHANICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
INDIA

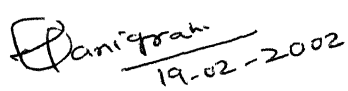
February, 2002



CERTIFICATE

It is certified that the work contained in the thesis entitled "***Parallelisation of a 3-D finite volume incompressible Navier-Stokes solver for complex geometry.***", by ***Mr. Abir Banerjee*** , has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. V. Eswaran
Dept. of Mechanical Engineering
I.I.T. Kanpur 208016



Dr. P. K. Panigrahi
Dept. of Mechanical Engineering
I.I.T Kanpur 208016

February, 2002

Abstract

In this work an existing semi-explicit, 3-D finite volume, incompressible Navier-Stokes fluid flow code is parallelised on a parallel computing platform comprising 9 PCs on a Linux environment with **ANULIB** as the message passing software. The PCs were Pentium-III, 866 MHz, 256 MB RAM and 20 GB HDD all connected by a 100 Mbps Fast Ethernet switch. ANULIB was uesd as the message passing library to take care of interprocessor communication.

The geometry taken was of a model chemical vapour deposition (CVD) reactor with two different substrate geometry both in 2-D and 3-D. Grids were both orthogonal and non-orthogonal. This type of finite volume algorithm has two steps: one predictor the other corrector. The equation-coupling in this algorithm reduces to a Poisson equation and is solved iteratively in the corrector step. In the parallel code both predictor and corrector steps are parallelised. The Poisson equation in the corrector step is solved by Jacobi iteration. Skin friction coefficient and velocity vectors were matched with the sequential results to confirm the validity of the parallel-computed results. Speed-up and efficiency parameters were measured for varying number of grids and on varying number of processors also.

The results show that satisfactory speed-up is obtained in the parallel environment, as much as 97% efficiency (=speed-up$\times$100/no. of processors) for the most computationally intesive problem attempted here.

ACKNOWLEDGEMENT

I acknowledge my sincere gratitude and thanks to my supervisors Dr. V.Eswaran and Dr. P.K.Panigrahi for their thoughtful guidance and invaluable suggestions. Their confidence and faith in me and my pursuit, have pushed me forward to work hard.

I have used the parallel processing facility at Dr. K.Muralidhar's laboratory. His constant inspiration and regular guidance made my endeavour easy going. I am grateful to him.

I thank my labmates Arnab, Kamlesh, Amit, Vivek for providing a good humorous environment in the laboratory. During the days of my final runs they have compromised a lot for me. I specially thank Arnab for helping me a lot in writing the 2nd chapter on finite volume formulation.

I extend my sincere gratitude to Mr. Jyotirmay Banerjee, Mr. Atul Sharma, Mr Sushanta Dutta, Mr. Andalib Tariq and Mr. Sunil Punjabi for their encouragement and helps during my thesis work.

I am pleased to mention the names of my hostel friends, Shamik Sen, Shamik Choudhury, Pradipta, Rakesh, Thathagata, Seresta, Debadi and Suman for thier encouraging friendship and generous help.

I am very much thankful to my brother Parag. He has helped me a lot in various occasions. I am thankful to my parents for their encouragement in taking my masters program. I thank my eldest brother and sister-in-law, for they have dealt with my sentiments with lot of care and patience.

I especially thank Dr. M.Verma. Her judicious medication has kept my nasal problem minimal during my stay in IIT.

At last I would like to thank all those people working silently to keep-up the greenery of this campus. Nothing rejoices me better than a green look of mother nature.

Abir Banerjee

Contents

List of Figures

Chapter 1

Introduction

Fluid mechanics and heat transfer have been intimately associated with computing for last forty years. Initially, heat transfer and fluid mechanics computations emphasized the evaluation of Fourier series solutions to conduction and internal flow problems. Later with the availability of commercial computers, the emphasis in both the areas has changed to finite difference/finite volume/finite element modeling of partial differential equations. The solution procedure of these methodologies are iterative in nature. The rapid improvement in both numerical techniques and computer technology carried forward the paradigm of computational fluid dynamics(CFD). Computational scientists and engineers routinely simulate on computers today phenomena too complex to be reliably predicted by theory and too large or expensive to be reproduced in the laboratory. Some of them are discussed below in brief.

1.1 Scope of CFD

- **Weather forecasting:** Weather forecasting is a classic example of the useful integration of up-to-date numerical methods and high-end computer technology. To forecast weather on a computer requires the solution of a general circulation model equations in a spherical coordinate system. A three-dimensional grid partitions the atmosphere by altitude, latitude, and longitude. The solution is also unsteady in nature. Given a grid with 270 miles on a side and an appropriate time increment, about 100 billion operations must be performed to compute a 24-hour forecast. This can be done in about 100 minutes on a computer capable of performing 100 million

operations per second.

- **Computational Aerodynamics:** Wind tunnel experiments have a number of fundamental limitations. This include the model size, wind velocity, density, temperature, wall interference, and other factors. Numerical simulations have none of these limitations. The replacement of wind tunnels testing has steadily increased along with the processing speed and memory capacity of computer being used. In many cases CFD has led to alternative designs and new configurations which improve performance of both commercial and defence airplanes.

- **Automotive Application:** CFD is being widely used as an efficient design tool in the automotive industry. Not only it is being used to optimize the outer shape from the point of view of minimizing both pressure and skin friction drag, but designs are continually being improved using CFD modeling. The combustion phenomenon in engines are simulated on computer and better understanding of both fluid mechanics and heat transfer phenomena is leading to fuel-efficient, less shock-creating and more environment friendly engines. It has greatly reduced the cost of making prototypes and carrying on hazardous experiments with them.

- **Biomedical Application:** Many people predict this century is going to be a century of bio-engineering. Likewise it is predicted that future of CFD lies in its success in biofluid mechanics. It is an emerging field of CFD. It mainly deals with arterial and respiratory fluid mechanics and cardiovascular modeling. Arterial transport phenomena are important to the understanding of vascular diseases. Of particular interest is the transport of macro-molecules(albumin, globulin) and dissolved gases(oxygen and carbon-dioxide) and through the arterial wall. The mathematical description of the flow is pulsatile, time dependent, three dimensional, incompressible generalized Navier-Stokes equations for a non-Newtonian inelastic fluid in case of blood flow through arteries and Newtonian fluid in case of air through air passages.

- **Other Applications:** Another classic examples of application of CFD includes flow simulation through gas turbines, exploration and recovery

simulations in the oil industry and defence applications. In all the fields CFD has given the proof of its tremendous potential and possibilities, and thus reaching out to newer fields.

1.2 Need of Powerful Computers

The cases discussed above demand raw computing power. This sheer need of megaflop and gigaflop level of computing power has led to the development of Supercomputers. Nearly 90 percent of a Supercomputer's calculational effort is spent in millions of floating operations per second termed MFLOPS(megaflops). A megaflop rating is qualified by stating the precision to which the operations are performed, i.e. number of significant digits. Supercomputers are generally designed to process 64 bit words keeping in mind the accumulation of round-off error associated with the many thousands of iterations which occur during solution.

1.2.1 Why Parallel Computing?

For those who compute in large scale, the quest for more powerful computing capability is addictive. Scientists and engineers are continually pursuing the limits of computing in their efforts to simulate nature in finer and finer details.For example to model the fine structure of 3-D turbulent flow over even the simplest of geometric forms requires an enormous number of computing cycles. The Supercomputers are bounded by the speed of light, i.e. the time for a signal to travel from one part of computer to another. Hence conventional super-computing approaches are limited at best to few billion instructions per second, which is not sufficient as seen in the above cases. Overcoming the sequential bottleneck requires new approaches of computing. The most logical means is to use many processors which work on simultaneously on the same problem, i.e. parallel computing. To pull a bigger wagon it is easier to add more oxen than to grow a gigantic ox.

1.2.2 Popularity of Desktop Networks

Several factors have stimulated the evolution of parallel computers. It is not only the speed of light and effectiveness of heat dissipation that limits on the speed of a

single computer, but also the prohibitive cost to develop advanced special-purpose hardware. It has been experienced that such systems are also different to program, and unable to track rapid improvements in the underlying technologies. At the same time clusters of workstations and even desktop computers connected by high-speed local area networks(LAN's) are gaining acceptance as a viable architecture for implementing high performance applications. This is for good reasons : desktop computers are affordable, unobiquitous, and track rapid improvements in microprocessor technology; LAN performance is becoming competitive with special-purpose multiprocessor interconnections; and clusters are easy to grow incrementally. And price-to-performance ratios become really favorable if the required computational resources can be found instead of purchased. This factor has caused many many to use existing workstations/desk-tops networks, originally purchased to do modest computational chores, as parallel computers by utilizing a workstation/desk-top network. This scheme has proven so successful, and the cost-effectiveness of individual workstations/desk-tops has increased so rapidly, that networks of desk-tops have been dedicated to parallel jobs that used to run on more expensive Supercomputers. Thus considerations of both peak performance and price/performance are pushing large scale computing in the direction of parallelism.

1.2.3 Message Passing versus Shared Memory Model

Parallel computers have been built differing in how they access each others memory. In shared memory models there is a shared global memory that can simultaneously be accessed by all the processors of the machine. The other model is such that each of the processor has its own memory that is exclusively local to itself, but communicate to each other by an interconnection network. This model is called the message passing model. It's difficult (and expensive) to make "true" shared-memory machines with more than a few tens of processors. One thing is clear that these shared-memory models have to be developed exclusively. But when we have some desktop computers available, it is just a matter of connecting them with a proper network to develop a formidable message passing parallel computer. Data is exclusively sent and received from the source to destination processors. This factor has led to the supremacy of message passing models over

shared-memory models, even though programming is easier in shared-memory models and inter-processor communication is costlier in message passing models. While debuggers for parallel programs are perhaps easier to write for the shared-memory model, it is arguable that the debugging process is itself easier in the message-passing paradigm. This is because one of the most common causes of error is unexpected overwriting of memory. The message model by controlling memory references more explicitly than any of the other model (only one process has direct access to any memory location), makes it easier to locate erroneous memory reads and writes.

1.3 Introduction to CVD

Chemical vapour deposition (CVD) is one of the most common process for manufacturing crystals. It refers to the formation of a crystalline material on a substrate by the reaction of chemicals in the gas phase across an activation energy barrier.

A large number of crystalline materials are now produced using CVD. In microelectronics manufacturing, for example, CVD is used to provide highly uniform thin layers (0.01-10μm) of semiconductors such as epitaxial silicon and gallium arsenide, dielectric such as silicon dioxide and silicon nitride, and metallic conductors W, Al and heavily doped polysilicon. Other applications include the production of metal thin flims such as Al_2O_3, TiC, SiC, B_4C and TiB_2 for hard coatings; anti-corrosive coating of BN, $MoSi_2$ for turbine blades and powders (Si_3N_4, SiC) to fabricate complicated parts via sintering and hot processing.

Compared to other film formation techniques (for example physical vapour deposition (PVD), electro deposition, liquid-base epitaxy and vapour-phase epitaxy), CVD offers a number of unique advantages such as versatility, quality , reproducability, and cost effictiveness. Other desirable features of CVD are its ability to provide conformal deposition and its relatively higher throughput. As a result, CVD has emearged as the dominant technology for producing films, especially in microelectronic applications, lasers and sensors.

The CVD process can be classified according to the method used to supply the activation energy for reactions. In *thermally activated* CVD, thermal energy is used for producing the gas-phase and surface reactions that result in the formation

of a thin film on a substrate. *Metal-organic* CVD is a special form of CVD, in which at least one of the reactants is a metal-organic compound. At the present time this is used for growing epitaxial films of compound semiconductors. In *plasma enhanched* CVD, a glow discharge plasma produced in gaseous reactants supplies much of the energy for the reaction. *Photo*-CVD utilizes photons for the excitation of reactant gases and gas species are absorbed on the substrate. However, a majority of the CVD process are thermally activated. The reactor in the present work is a thermally activated one.

1.4 Objective of the Present Work

Finite volume methods are well known in solving fluid flow problems involving complex geometry. Existing methods which solve the incompressible Navier-Stokes equations fall into categories, namely, semi-explicit and implicit schemes. In the first the momentum equations are discretized in an explicit manner with the exception of the pressure gradient terms, which are treated implicitly; the continuity equation is also enforced implicitly. As a consequence, the equation-coupling reduces to a Poisson equation for the pressure corrections.

It has been experienced that the development of such codes and their evolution must face three problems :

- the complexity of the physics,
- the complexity of the numerics,
- the large computing time.

The objective of the present work is to overcome the large computing time barrier for a finite volume incompressible Navier-Stokes code, employing the semi-explicit scheme. This code was developed by Eswaran[22]. It has been tested on a wide variety of geometry (both simple and complex) and wide range of solution schemes yielding very well established results. Its flexibility is such that very little change is required to accommodate a new geometry.

Keeping in mind the versatility and wide applicability of this code, it was aimed to reduce large computational time using massively parallel computers on distributed memory architecture.

The model problem taken up for parallelisation is the flow simulation through a CVD reactor with two different geometries in both 2-D and 3-D (its geometry is discussed in the next chapter). From our past experience we know of its large computational time. As much of the computations are used to solve the (elliptic) Poisson equation for the pressure corrections, the code is not *ideally* suited for parallelization from a technical sense. However it is our endeavour to see how much benifits accrue due to parallelization. The code has been parallelised with the following aims at mind.

- The solution algorithm be kept untouched, except for the solver (previously it was Gauss-Seidel, for our case Jacobi). Data parallelism in the algorithm has to be exploited.
- The communications should be irrespective of the geometry of the physical domain.
- Communication time should be as low as possible.
- Accuracy in results should be undoubted, and the results obtained the same as would be on a serial computer.

Chapter 2

Description of Model Problem and its Finite Volume Formulation

2.1 Description of the Model Problem

Flow over a CVD reactor is simulated in our present work. The geometry of the problem is cylindrical. At the entrance of the reactor there is a central jet surrounding which there are four perpherial jet at a 90 degree intervals. The exit of the pipe is fully open. In between there is a block called the "substrate" .

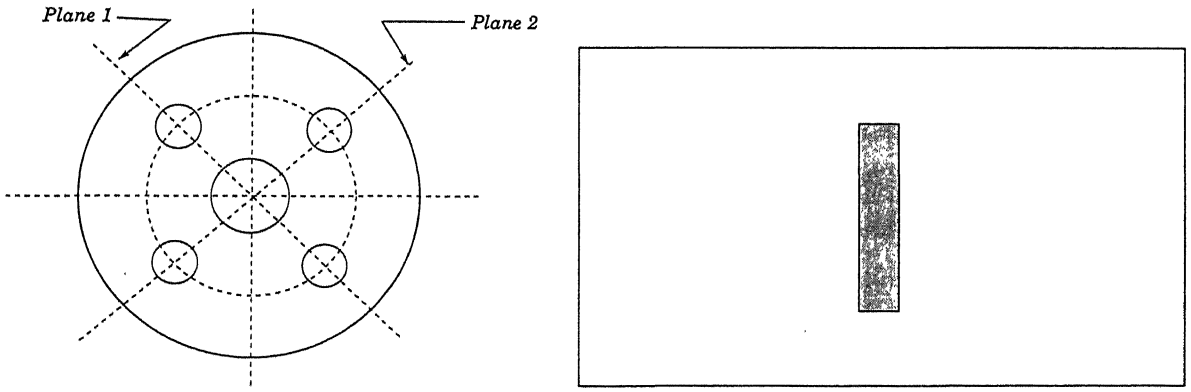


Figure 2.1: A CVD reactor with staright block.

In our case the block position was taken to be fixed and at an approximately one diameter distance away from the entrance. On this substrate, deposition is supposed to take place. Two geometry as for the substrate was considered, one a straight circular block and the other one is a hemispherical block. To generate grid for the curved block configuration orthogonal grid generation technique was

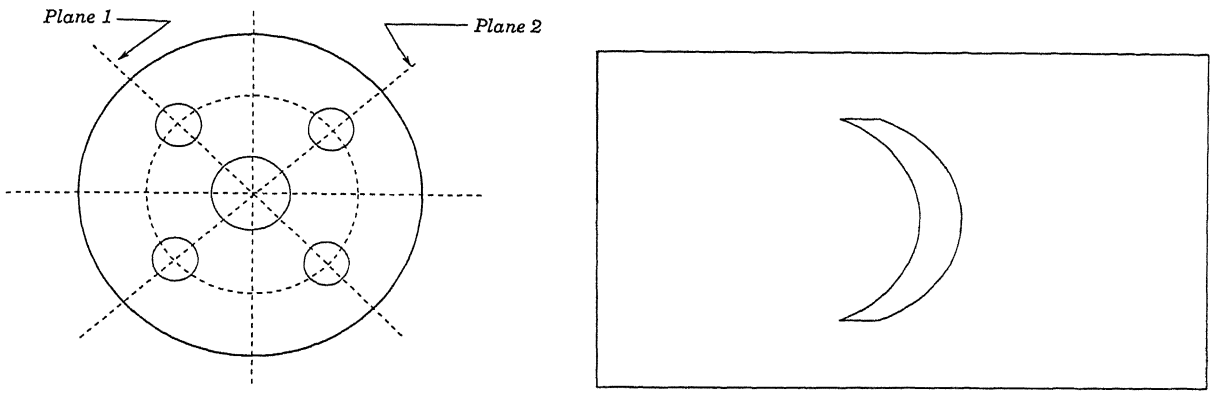


Figure 2.2: A CVD reactor with curved bolck.

used. This grid has been generated as a part of M.Tech thesis of Mr. Arnab Kumar De. However, the parallel computations for the 3-D curved block case could not be completed and is not presented here.

The ultimate aim is predict the deposition of zinc sulphide on the block surface during the CVD process.To achieve it numerically , first step is to obtain the sulution of momentum equations. Then only energy and mass-transfer equations can be solved. Zinc sulphide deposition is achieved through gas-phase recation of zinc vapour and hydrogen-sulphide gas. The central jet carries H_2S, as Argon being the carrier gas, and through the four peripherial jets zinc vapour is injected, again with Argon. The reactor chamber is partially evacuated.

This central and peripheral jat arrangement enhances mixing of the two incoming species facilitating the reaction that produces zinc sulphide. The flow field ultimately dictates the path of the species (ZnS) generated. So species concentration field along with velocity field predicts the species deposition rate. The concentration field of the species is also dependent on the velocity field.

2.2 Solution of Momentum equations

In fluid flow problems solution of the momentum equations constitute the most important steps. In incompressible flow due to non-availability of any explicit relationship for pressure complicates the situation. In most of the solution method used today implement some kind of pressure-velocity coupling to get velocity field which nearly explains the pressure field. Here we present the finite volume formulation, solution methodology and certain issues which influence the solution

algorithm. Semi-explicit type of time marching is adopted. Calculations have been done on physical geometry itself instead of in generalized coordinate system. This is justified because in transformed domain the governing equations become quite cumbersome and essentially more computational efforts are required. We have worked with the Cartesian velocities. Variable arrangement have been done in non-staggered fashion. To avoid velocity pressure decoupling associated with calculations in non-staggered grids momentum interpolation has been used.

In the present study the momentum equations were solved with following assumptions, with proper justifications behind them.

1. The fluid is incompressible. This is reasonable because flow rates as well as the flow velocities are quite small.
2. The flow is everywhere laminar.
3. The governing equations are integrated in time, but the focus is only on the steady state. This is justified because it has been observed from experiments that the veloity field get established within a few secondes.
4. Buoyancy effects are negligible. Thus other two transport equations are decoupled from the velocity transport equation. Similarly viscous dissipation is also negligible.

2.2.1 Governing equations

All the transport equations we encounter are basically the generic advection diffusion equation. The Navier Stokes equation for laminar incompressible flow for an arbitrary control volume in physical domain is written in integral form:

$$\frac{\partial}{\partial t}\int_v \rho dV + \int_s \rho \overline{u}.\overline{dS} = 0 \tag{2.1}$$

$$\frac{\partial}{\partial t}\int_v \rho\phi dV + \int_s (\rho\overline{u}\phi - \Gamma_\phi \nabla\phi)\,.\,\overline{dS} = \int_v S_\phi dV \tag{2.2}$$

where ρ represents fluid density, ϕ stands for any extensive property which is transported. S_ϕ is the volumetric source term. For incompressible flow the above equations takes the form:

$$\int_s \rho\overline{u}.\overline{dS} = 0 \tag{2.3}$$

$$\frac{\partial}{\partial t}\int_v \phi dV + \int_s (\overline{u}\phi - \frac{\Gamma_\phi}{\rho}\nabla\phi)\,.\,\overline{dS} = \frac{1}{\rho}\int_v S_\phi dV \tag{2.4}$$

In present calculations for the three momentum equations $\overline{u}, \overline{v}, \overline{w}$ take the place of ϕ.

2.2.2 Description of the Finite Volume

The solution domain is divided into a number of contiguous finite volumes. The control volumes are defined by the coordinates of their vertices. In complex geometry these control volumes are formed by joining their eight vertices by straight lines. The coordinates of the control volumes are generated by Grid Generation technique. Collocated grid system is used in which all the dependent variables ($\overline{u}, \overline{v}, \overline{w}$, p) are defined at the centroid of the control volumes. Figure 2.1 shows a typical finite volume employed in computations.

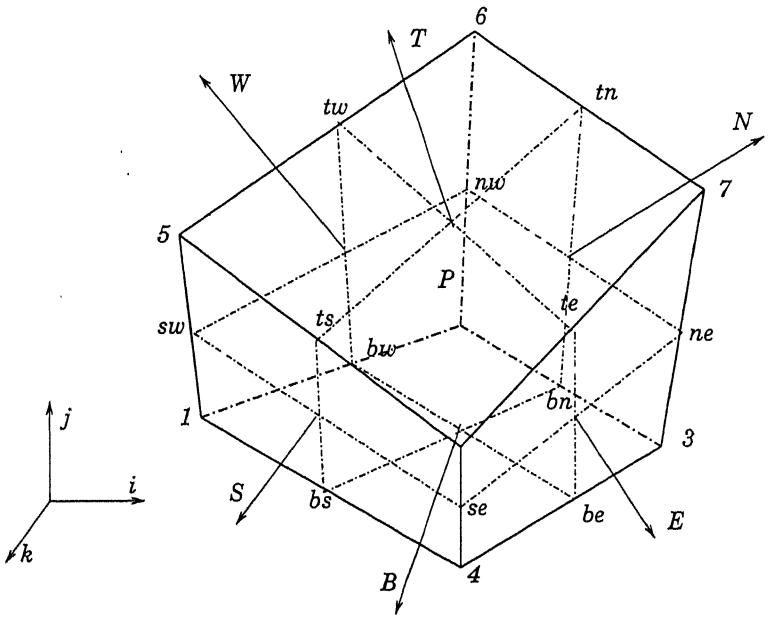


Figure 2.3: A three-dimensional arbitrary control volume.

The six neighbouring control volume centers are denoted by E, W, N, S, T, B for the east, west, north, south, top, below control volumes respectively. The cell face centres are denoted by e, w, n, s, t, b in the same order. Edge centres are te, tn, tw, ts, be, bn, bw, bs, ne, nw, sw, se as shown in the figure 2.1. For the computations we need values of cell face surface area and volume of the cells. The surface area for each of six faces of the elemental volume and its volume have been calculated using method suggested by Kordula and Vinokur[3].

2.3 Discretization Procedure

Here we present the techniques used to discretize various integrals of the continuity and momentum equation.

2.3.1 Continuity Equation

the integral form of the continuity equation is given by

$$\int_s \rho \mathbf{u} \,.\, \mathrm{dS} \approx \sum \rho(\mathbf{u}.\mathbf{S})_j = \sum \rho(\mathbf{u}_j.\mathbf{S}_\mathrm{j}) = \sum F_j \tag{2.5}$$

where F_j is the outward mass flux through face j, defined by

$$F_j = \rho \mathbf{u}_j.\mathbf{S}_j \tag{2.6}$$

Thus discretized continuity equation states that total mass flux through all the cell faces vanish and in turn net mass in a control volume remains the same which is nothing but the principle of conservation of mass. The discretized continuity equation is given by

$$\sum F_j = F_w + F_s + F_e + F_n + F_t + F_b = 0 \tag{2.7}$$

2.3.2 Momentum equations

The momentum equation when written in generic advection diffusion equation it contains four terms as unsteady, convective, diffusive and source integrals. Here we present the discretization method we used for all the terms

Unsteady term

The unsteady term in the governing equation is discretized with the help of the basic assumption that all the dependent variables for which we seek solution are defined at the cell centroid. Thus if follows

$$\frac{\partial}{\partial t}\int_v \rho\phi dV \approx \frac{(\rho\phi V)_P^{n+1} - (\rho\phi V)_p^n}{\Delta t} \approx \rho V \frac{\phi_P^{n+1} - \phi_P^n}{\Delta t} \tag{2.8}$$

where V is the volume of the control volume and Δt the time increment for time marching. The time discretization done above is first order accurate.

Convective term

The convective integral in the governing equation is approximated as following

$$\int_s \rho \mathbf{u}\phi \,.\, \mathbf{dS} \approx \sum (\rho \mathbf{u} \,.\, \mathbf{dS})_j \phi_j = \sum F_j \phi_j \tag{2.9}$$

where ϕ_j is the value of the dependent variable at cell face and F_j is the mass flux through the face j. Thus the discretized equation is given by

$$\int_s \rho \mathbf{u}\phi \,.\, \mathbf{dS} \approx F_w\phi_w + F_s\phi_s + F_e\phi_e + F_n\phi_n + F_t\phi_t + F_b\phi_b \tag{2.10}$$

where ϕ_i is the value of dependent variable at the face i. As all the variables are defined at the cell centroid we have to use some interpolation scheme to obtain value of the same at the face centre. In the present work this has been accomplished by taking contribution of both second order central difference and first order upwind difference scheme. Here we demonstrate the two scheme applied in the present work and then the form they take when mixed. Central difference is applied to calculate the face centre value of the dependent variable in the following way

$$\phi_j = \frac{V_J}{V_J + V_{J+1}}\phi_{J+1} + \frac{V_{J+1}}{V_j + V_{J+1}}\phi_J \tag{2.11}$$

where j and J represent the face centre and cell centre value. Here J and $J+1$ are two neighbouring cells adjacent to the face j. It is to be noted that for uniform grids the above formula gives second order accuracy but for non uniform grids this volume weighted formula gives only first order accuracy. The upwind scheme rightly manifests transportive property of the fluid flow equations in the local flow direction but it introduces numerical diffusion. This scheme is written as following

$$F_j\phi_j = |F_j,\, 0|\phi_J + |-F_j,\, 0|\phi_{J+1} \tag{2.12}$$

Thus by this scheme dependent variable at the face centre takes the value of the downstream cell if mass flux is positive and it takes the value of upstream cell if

mass flux is negative. Now the medley of the two schemes takes the form

$$F_j\phi_j = (|F_j,\,0|\phi_J + |-F_j,\,0|\phi_{J+1}) + \gamma\,[F_j(\frac{V_J}{V_J+V_{J+1}}\phi_{J+1} + \frac{V_{J+1}}{V_j+V_{J+1}}\phi_J) - (|F_j,\,0|\phi_J + |-F_j,\,0|\phi_{J+1})] \tag{2.13}$$

with the factor γ signifies the contribution of each scheme. It varies between 0 and 1, with 0 implies fully upwind and 1 implies fully central difference scheme. There is no universal rule to choose this factor and it depends on the nature of a problem, numerical technique being used, range of parameters taken into account etc. to name a few. In the present study we use 0.5 as its value for all the cases as we are only motivated in the low Reynolds number region. Following in the same line we calculate the convection flux at all the faces of a control volume. For example at the east face of any cell:

$$F_e\phi_e = (|F_e,\,0|\phi_P - |-F_e,\,0|\phi_E) + \gamma\,[F_e(\frac{V_E}{V_E+V_P}\phi_P + \frac{V_P}{V_E+V_P}\phi_E) - (|F_e,\,0|\phi_P - |-F_e,\,0|\phi_E)] \tag{2.14}$$

Diffusion term

The diffusion flux of variable ϕ as occurred in the governing equation can be approximated in the following way

$$\int_s \Gamma_\phi \nabla\phi \,.\, \mathrm{dS} \approx \sum (\Gamma_\phi \nabla\phi \,.\, \mathrm{S})_j = \sum F_j^d \tag{2.15}$$

Diffusion discretization has been done in such a way that it can be fairly applied to complex geometries with curvilinear grids. Diffusion mechanism is considered in both normal and tangential direction which are termed as normal diffusion and cross diffusion. For any face outward normal surface vector can be written as the linear combination of three linearly independent unit vectors which are not necessarily orthogonal. Thus

$$\mathbf{S}_j = \alpha_1\mathbf{n}_1 + \alpha_2\mathbf{n}_2 + \alpha_3\mathbf{n}_3 \tag{2.16}$$

where $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$ are linearly independent unit vectors given by in terms of their Cartesian components as

$$\mathbf{n}_1 = (n_{11}\, n_{12}\, n_{13}) \tag{2.17}$$

$$\mathbf{n}_2 = (n_{21}\, n_{22}\, n_{23})$$

$$\mathbf{n}_3 = (n_{31}\, n_{32}\, n_{33})$$

Thus the vector product can be written as

$$\nabla\phi \,.\, \mathbf{S} = \nabla\phi \,.\, (\alpha_1\mathbf{n}_1 + \alpha_2\mathbf{n}_2 + \alpha_3\mathbf{n}_3 \tag{2.18}$$

$$= \alpha_1\nabla\phi \,.\, \mathbf{n}_1 + \alpha_2\nabla\phi \,.\, \mathbf{n}_2 + \alpha_3\nabla\phi \,.\, \mathbf{n}_3$$

Now if $\Delta\phi_1$, $\Delta\phi_2$, $\Delta\phi_3$ are the differences in ϕ along the three line segment $\Delta\mathbf{x}_1$, $\Delta\mathbf{x}_2$, $\Delta\mathbf{x}_3$, then

$$\Delta\phi_1 = \nabla\phi \,.\, \Delta\mathbf{x}_1, \quad \Delta\phi_2 = \nabla\phi \,.\, \Delta\mathbf{x}_2, \quad \Delta\phi_3 = \nabla\phi \,.\, \Delta\mathbf{x}_3 \tag{2.19}$$

If $\Delta\mathbf{x}_1, \Delta\mathbf{x}_2, \Delta\mathbf{x}_3$ are in the direction of $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$ and $\Delta x_1, \Delta x_2, \Delta x_3$ be the magnitudes of the line segments respectively then

$$\Delta\phi_1 = \nabla\phi \,.\, \mathbf{n}_1\Delta x_1, \quad \Delta\phi_2 = \nabla\phi \,.\, \mathbf{n}_2\Delta x_2, \quad \Delta\phi_3 = \nabla\phi \,.\, \mathbf{n}_3\Delta x_3 \tag{2.20}$$

and so

$$\nabla\phi \,.\, \mathbf{n}_1 = \frac{\Delta\phi_1}{\Delta x_1}, \quad \nabla\phi \,.\, \mathbf{n}_2 = \frac{\Delta\phi_2}{\Delta x_2}, \quad \nabla\phi \,.\, \mathbf{n}_3 = \frac{\Delta\phi_3}{\Delta x_3} \tag{2.21}$$

So the diffusion flux at any face becomes

$$F_j^d = \Gamma_\phi\nabla\phi \,.\, \mathbf{S}_j = \Gamma_\phi(\alpha_1\frac{\Delta\phi_1}{\Delta x_1} + \alpha_2\frac{\Delta\phi_2}{\Delta x_2} + \alpha_3\frac{\Delta\phi_3}{\Delta x_3}) \tag{2.22}$$

Now equation (2.16) can be written in matrix form as

$$\begin{bmatrix} n_{11} & n_{21} & n_{31} \\ n_{12} & n_{22} & n_{32} \\ n_{13} & n_{23} & n_{33} \end{bmatrix} \begin{Bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{Bmatrix} = \begin{Bmatrix} S_{1j} \\ S_{2j} \\ S_{3j} \end{Bmatrix} \tag{2.23}$$

where S_{1j}, S_{2j}, S_{3j} are the three Cartesian components of the outward normal surface vector $\mathbf{S}_j$, This matrix can be inverted to get the values of α_1, α_2, α_3

We have used Cramer's rule to invert the matrix, which is given by

$$\alpha_1 = \frac{D_1}{D}, \quad \alpha_2 = \frac{D_2}{D}, \quad \alpha_3 = \frac{D_3}{D} \tag{2.24}$$

where D is the determinant of the coefficient matrix of the equation (2.23). The matrices D_1, D_2, D_3 are given by replacing respective columns by the left hand side vector

In the diffusion discretization there are two parts, one implies normal diffusion and the other implies cross diffusion. It can be shown that for orthogonal uniform grids the cross diffusion term vanishes but for non-uniform curvilinear grids it strengthens the diffusion mechanism as depicted by the numerical treatment.

Clearly for cross diffusion we need the value of the dependent variable at the edge centres. These values have been calculated by the volume weighted interpolation technique using cell centre values of four neighbouring cells of an edge. Values at the four edge centres on east face is written below.

$$\phi_{te} = \frac{V_{TE}\phi_P + V_E\phi_T + V_P\phi_{TE} + V_T\phi_E}{V_{TE} + V_E + V_P + V_T} \tag{2.25}$$

$$\phi_{se} = \frac{V_{SE}\phi_P + V_S\phi_E + V_P\phi_{SE} + V_E\phi_S}{V_{SE} + V_S + V_P + V_E} \tag{2.26}$$

$$\phi_{be} = \frac{V_{BE}\phi_P + V_B\phi_E + V_P\phi_{BE} + V_E\phi_B}{V_{BE} + V_B + V_P + V_E} \tag{2.27}$$

$$\phi_{ne} = \frac{V_{NE}\phi_P + V_N\phi_E + V_P\phi_{NE} + V_E\phi_N}{V_{NE} + V_N + V_P + V_E} \tag{2.28}$$

Calculations for the edge centre values at other faces are straightforward and are done in the same way as described above.

Pressure Gradient term

In the momentum equation pressure term is treated explicitly in the predictor step. Its discretization after integrating over a finite volume is written below for the i-th momentum equation

$$\int_v \nabla p \,.\, \mathbf{n}_i dV = \int_v \nabla p \,.\, \mathbf{n}_i dV + \int_v p\nabla \,.\, \mathbf{n}_i dV \tag{2.29}$$

$$= \int_v \nabla \,.\, (p\mathbf{n}_i) dV$$

$$
\begin{aligned}
&= \int_s p\mathbf{n}_i \,.\, \mathrm{dS} \\
&= \sum p_j \mathbf{n}_i \,.\, \mathbf{S}_j \\
&= \sum p_j \, \mathbf{S}_{ij}
\end{aligned}
$$

This term is similar to the diffusion flux, so we have discretized it in the same way as for the diffusion term.

2.3.3 Time integration Scheme

The momentum equation is mathematically mixed type in nature. It can be visualized as elliptic in space and parabolic in time. So time marching strategy with satisfying the boundary conditions as accurately as possible in each time step has been adopted as the solution methodology. The momentum equation has been time integrated semi explicitly with each term treated explicitly with the exception of the pressure term which is treated implicitly. The time integration consists of two steps. In the first step which is the predictor step velocities are predicted with pressure being taken explicitly and in the second step we enforce continuity equation implicitly and the momentum equation also written with the pressure term as implicit. Thus pressure velocity coupling reduces to a Poisson equation for pressure correction. This equation has been solved iteratively until the fluid flow domain becomes divergence free. Explicit formulation does not involve any matrix inversion for the velocity calculation. However explicit formulation always suffer from time step restriction. Moreover due to very small time step minute transient details are expected from an explicit calculation. Although the present study has not been directed towards getting transient details the issues which are mentioned above justifies the choice of the explicit algorithm. The predictor and the corrector step with the equations to solved in each step is written below

The Predictor step

In the predictor step velocities are predicted using values from current time step which is known already as

$$\rho V_p \frac{\mathbf{u}_P^* - \mathbf{u}_P^n}{dt} + \sum (F_j^c + F_j^d)^n = -\sum p_j S_{ij} \tag{2.30}$$

Which leads to the predicted velocities as

$$\mathbf{u}_p^* = \mathbf{u}_p^n - \frac{dt}{\rho V_P}(\sum(F_j^c + F_j^d)^n - \sum p_j^n S_{ij} \tag{2.31}$$

The Corrector step

By semi explicitly advancing in time we get the equation

$$\mathbf{u}_p^{n+1} = \mathbf{u}_p^n - \frac{dt}{\rho V_P}(\sum(F_j^c + F_j^d)^n - \sum p_j^{n+1} S_{ij} \tag{2.32}$$

By subtracting equation (2.32) from equation (2.31) we get

$$\mathbf{u}_p^{/} = -\frac{dt}{\rho V_P} \sum p_j^{/} S_{ij} \tag{2.33}$$

where $\mathbf{u}_p^{/}$ and $p^{/}$ velocity and pressure correction to be done on predicted velocities and guessed pressure, is given by

$$\mathbf{u}_p^{/} = \mathbf{u}_p^{n+1} - \mathbf{u}_p^* \quad and \quad p^{/} = p^{n+1} - p^n \tag{2.34}$$

If we cast the above equation into the integral form we get

$$\begin{aligned} \int_v \mathbf{u}^{/} dV &= -\frac{dt}{\rho} \sum p_j^{/} \mathbf{n}_i \,.\, \mathbf{S}_j \\ &= -\tfrac{dt}{\rho} \textstyle\int_s \nabla.(p^{/}\mathbf{n}_i) dV \\ &= -\tfrac{dt}{\rho}(\textstyle\int_v \nabla p^{/}.\mathbf{n}_i dV + \int_v p^{/} \nabla.\mathbf{n}_i dV) \\ &= -\tfrac{dt}{\rho} \textstyle\int_v \nabla p^{/} dV \end{aligned} \tag{2.35}$$

With the integration being done over an arbitrary finite volume, the integrand can be pulled out of the integral to get

$$\mathbf{u}^{/} = -\frac{dt}{\rho} \nabla p^{/} \tag{2.36}$$

In this step we enforce continuity implicitly, thus from equation (2.7)

$$\sum F_j^{n+1} = 0$$

Writting this term as sum of predicted flux based on predicted velocities in first step and the amount of flux to be corrected to get a divergence free flow field

$$\sum F_j^* + \sum F_j^{/} = 0 \tag{2.37}$$

This gives

$$\sum F_j^* = -\sum F_j^{/} \tag{2.38}$$

Thus equation (2.36) and equation (2.38) constitute the corrector step. These two equations have to be solved simultaneously and this has been accomplished by iterative technique. This two equations are reduced to a Poisson equation in pressure correction which is solved to get a divergence free velocity field.

Pressure Correction equation

It can be shown the two equations to be solved in the correcter step leads to a Poisson equation in pressure correction in the following way. The two terms in the Equation (2.38) when written in integral form takes the form

$$\begin{aligned} \sum F_j^* &= \int_s \rho \mathbf{u}^* . \mathbf{dS} \\ &= \rho \int_v \nabla . \mathbf{u}^* dv \end{aligned}$$

and

$$\begin{aligned} -\sum F_j^{/} &= -\int_s \rho \mathbf{u}^{/} . \mathbf{dS} \\ &= -\rho \int_v \nabla . \mathbf{u}^{/} dv \\ &= dt \int_v \nabla . (\nabla p^{/}) dV \\ &= dt \int_v \nabla^2 p^{/} \, dV \end{aligned}$$

Thus the integral form of the equation (2.38) is

$$dt \int_v \nabla^2 p^{/} \, dV = \rho \int_v \nabla . \mathbf{u}^* dV \tag{2.39}$$

As the integration has been carried out over an arbitrary finite volume, the integrand can be pull out of the integration to get

$$\nabla^2 p^{/} = \frac{\rho}{dt} \nabla . \mathbf{u}^* \tag{2.40}$$

So solving equation (2.36) is equivalent to solution of equation (2.40) which is the Poisson equation in pressure correction. The two equations to be solved simultaneously in the corrector step can also be reduced to a form which is more amenable to Jacobi or Gauss-Seidel iterative method. This is called residual form of the pressure correction equation. This form is obtained in the following way.

Mass flux correction can be written in terms of pressure correction by using equation (2.36)

$$\begin{aligned} F_j^{/} &= (\rho \mathbf{u}_j^{/}) \,.\, \mathbf{S}_j \\ &= -dt \nabla p^{/} \,.\, \mathbf{S}_j \end{aligned} \tag{2.41}$$

So equation (2.38) becomes

$$\sum F_j^{*} - \sum dt \nabla p^{/} \,.\, \mathbf{S}_j = 0 \tag{2.42}$$

Now the pressure correction term in the above equation. can be discretized in the same way as the diffusion term as they are similar in nature

$$dt \nabla p^{/} \,.\, \mathbf{S}_j = dt[(\sum \nabla p^{/} \,.\, \mathbf{S}_j)_{nd} + (\sum \nabla p^{/} \,.\, \mathbf{S}_j)_{cd}] \tag{2.43}$$

where the subscript nd and cd corresponds to normal diffusion and the cross diffusion terms explained in context of the diffusion flux. It is to be noted that $p^{/}$ in the neighbouring cells in the cross diffusion term take their current values, so this term contains values of both current and previous iteration levels. In short this term is represented by T_{cd}. The normal diffusion term is discretized in the same way as the diffusion flux. Using equation (2.22) we write

$$\begin{aligned} dt[(\sum \nabla p^{/} \,.\, \mathbf{S}_j)_{nd} = dt[&\tfrac{\alpha_1}{\Delta x_1} |_w \, (p_P^{/} - p_W^{/}) + \tfrac{\alpha_2}{\Delta x_2} |_s \, (p_S^{/} - p_P^{/}) + \tfrac{\alpha_1}{\Delta x_1} |_e \, (p_E^{/} - p_P^{/}) + \\ &\tfrac{\alpha_2}{\Delta x_2} |_n \, (p_P^{/} - p_N^{/}) + \tfrac{\alpha_3}{\Delta x_3} |_t \, (p_T^{/} - p_P^{/}) + \tfrac{\alpha_3}{\Delta x_3} |_b \, (p_P^{/} - p_B^{/})] \\ &= dt \sum \tfrac{\alpha}{\Delta x} |_{nb} \, p_{nb}^{/} - a_P \, p_P^{/} \end{aligned}$$

where nb denotes all the neighbours of the cell with centroid at P and the coefficient a_P is given by

$$a_p = -dt(\frac{\alpha_1}{\Delta x_1} |_w - \frac{\alpha_2}{\Delta x_2} |_s - \frac{\alpha_1}{\Delta x_1} |_e + \frac{\alpha_2}{\Delta x_2} |_n - \frac{\alpha_3}{\Delta x_3} |_t + \frac{\alpha_3}{\Delta x_3} |_b) \tag{2.44}$$

Thus equation (2.42) becomes

$$-\sum F_j^{*} + dt \, T_{cd} + dt \sum \frac{\alpha}{\Delta x} |_{nb} \, p_{nb}^{/} - a_P \, p_P^{/} = 0 \tag{2.45}$$

Writting the pressure correction at the cell with centroid at P

$$(p'_P)^{n+1} = \frac{-\sum F_j^* + dt\,T_{cd} + dt \sum \frac{\alpha}{\Delta x} |_{nb}\, p'_{nb}}{a_P}$$
$$= (p'_P)^n + \frac{-\sum F_j^* + dt\,T_{cd} + dt \sum \frac{\alpha}{\Delta x} |_{nb}\, p'_{nb} - a_P\,(p'_P)^n}{a_P}$$

Now the term $dt\,T_{cd} + dt \sum \frac{\alpha}{\Delta x} |_{nb}\; p'_{nb} - a_P\,(p'_P)^n$ is the discretized form of the corrective flux with pressure correction at the cell with centroid at P is at previous iteration step. This corrective mass-flux based on p'_P at the previous iteration level together with the predicted mass flux gives the residual ($\Re$) which is to be reduced to zero to get the divergence free velocity field. This residual is written as

$$\Re = -\sum F_j^* - \sum F_j' \tag{2.46}$$

$$\Re = -\sum F_j^* + dt\,T_{cd} + dt \sum \frac{\alpha}{\Delta x} |_{nb}\; p'_{nb} - a_P\,(p'_P)^n \tag{2.47}$$

Thus using equation.() we write the formula for the pressure correction as

$$(p'_P)^{n+1} = (p'_P)^n + \frac{\Re}{a_P} \tag{2.48}$$

This expression has been used to get pressure correction at the next iteration level.

2.3.4 Solution Algorithm

1. Initial conditions for velocities are given and pressure is guessed throughout the domain to start calculation.

2. Velocities are predicted in the prediction step using the initial condition or values from the previous time step.

3. Predicted values of mass flux is calculated based on predicted velocities and will be used in the corrector step.

4. Initial guessed values for pressure correction are assigned to start iteration in corrector step

5. Corrective mass flux, F_j' for all the cell faces j are calculated by equation (2.41) and residual at each cell, $\Re$ is calculated.

6. Pressure correction values, p' at all the cells are updated by equation (2.48) given by

$$p' \leftarrow p' + \frac{\Re}{a_P}$$

7. Go to step 5 until the value of the residual at each cell reduces to a pre-assigned small value.

8. Update mass flux at cell faces, velocity and pressure at cell centres by

$$p \leftarrow p + p', \quad \mathbf{u} \leftarrow \mathbf{u} + \mathbf{u}'$$

This concludes the calculations at a single time step.

9. Go to step 2 and continue the time marching until the difference in velocities between two consecutive time steps becomes smaller than a pre-assigned small value.

2.3.5 Initial and Boundary conditions

As our aim was to obtain steady state solution for the problem, we started with arbitrary guessed value as the initial condition. However to obtain meaningful transient variation initial condition should be given in accordance with the physical problem.

As all the dependent variables are defined at the cell centres and the boundaries lie on the cell faces, we assume a fictitious layer of cell in the adjacent to each boundary segment. By interpolating values between the fictitious cell and the real cell adjacent to a boundary we calculate the values at the corresponding boundary. It is important to note that if the above mentioned fictitious layer of cells are perfect image of the real cells adjacent to that boundary, boundary conditions are satisfied exactly. Due to the curvilinear non-uniform grids in the present study we have used volume weighted interpolation to set boundary conditions. In the present work we have used the following boundary conditions.

At Inlet plane

Specified velocity distribution is assigned for both the central and the peripheral jets. For all cases we have taken uniform velocity profile for both the jets.

At exit plane

Convective boundary conditions are used at the exit. The essence of this condition is that there are no significant changes in velocities in the streamwise direction.

At the Solid surfaces

No slip boundary conditions have been applied on all the solid surfaces except the corner cells of the substrate which are treated with certain care.

Corner cell treatment

The two corner cells of the solid obstacle is multiply connected with the flow domain. It is different from the rest of the boundaries in the sense that it has more than one face facing the fluid flow domain. In the present work we have set no mass penetration conditions at these cells.

At the centreline of the pipe

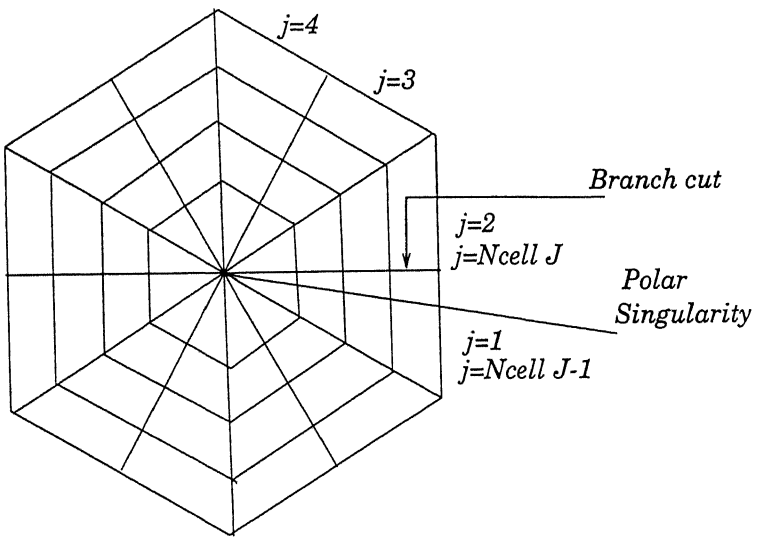


Figure 2.4: Polar singularity and branch-cut on a circular geometry.

For a circular geometry the grid points on the centreline collapse to a single point in the centre, causing what we call a *polar singularity.* However for the numerical solver, this grid line is a "boundary", and boundary conditions are required on it to obtain the solution.

Treatment of polar singularity

Centreline cells are assumed to be fictitious cells of zero volume and their surface areas are also zero. The boundary values in the zero volume cells are specified as the average of the cell-centre values from $j = 2$ to $j =$ NcellJ-1 of the first real cell at $k = 2$.

Branch-cut and its treatment

The branch-cut in the Figure (2.5) should be noted carefully. The branch-cut is introduced for computational purposes but across it all variables are continuous. One side of the branch-cut is north boundary of $j = 2$ cells and other side is south boundary of $j =$ NcellJ-1 cells. Fictitious cells are used to specify boundary conditions at these boundaries. As shown in figure (2.5) the fictitious cells at $j =$ 1 is exactly same as the cell at $j =$ NcellJ-1 and the fictitious cell at $j =$ NcellJ is same as the cells at $j = 2$. So the boundary condition for any variable ϕ across the branch-cut is:

$$\phi_{i,1,k} = \phi_{i,NcellJ-1,k} \tag{2.49}$$

$$\phi_{i,NcellJ,k} = \phi_{i,2,k} \tag{2.50}$$

Chapter 3

Parallelization Strategy

3.1 Introduction to Parallel Computing

Before we go into solving some real problem on a parallel computer it will be helpful to give a cursory look at parallel computers and its related taxonomical terms. We can classify parallel computer architectures by the concepts of instruction stream and data stream. An *instruction stream* is a sequence of instructions performed by a computer; a *data stream* is a sequence of data used to execute an instruction stream. Given the possible multiplicity of instruction and data streams, four classes of computers result.

Single-Instruction stream, Single Data stream (SISD). This is the traditional sequential computer consisting of one processor, memory and one communication channel as shown in Figure 1. obviously this computer architecture will always be restricted by the part of the three elements with least capacity. With the present technology providing almost unlimited memory and extremely fast central processing units, the bottleneck is the communication channel.

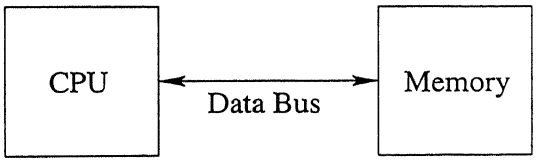


Figure 3.1: Schematic of a sequential computer.

Single-Instruction stream, Multiple Data stream (SIMD). Processor arrays fall into this category. A processor array executes a single stream of instructions, but contains a number of arithmetic processing units, each capa-

ble of fetching and manipulating its own data. This architecture is available in distributed and shared memory systems. The shared memory architecture is illustrated in Figure 2 and the distributed memory architecture in Figure 3. This architecture overcomes the above mentioned bottleneck in the SISD systems by implementing multiple processors and memory module/modules working on the single instruction set. Many large parallel computers use this principle, taking advantage of simplified programming task as the number of number of data sets are program independent and the same instruction set is performed on all processors at any given time(synchronously). This leads to data parallelism where the problem should be highly regular. Host communication may prove to be troublesome for this architecture since each processor must share the same communication path.

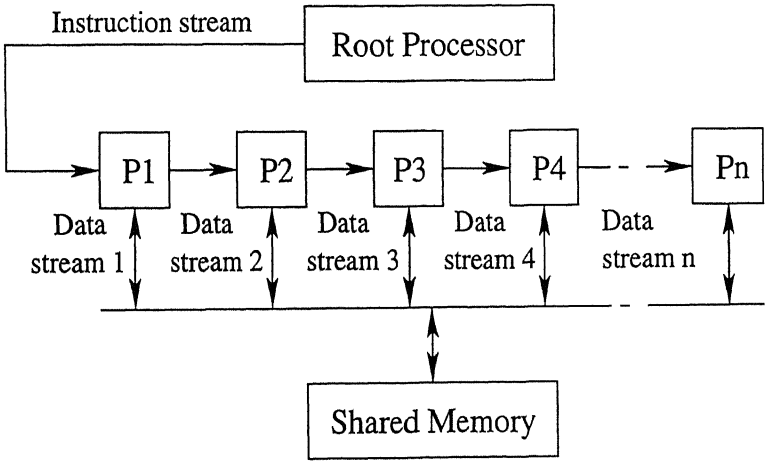


Figure 3.2: Schematic of a shared memory SIMD computer.

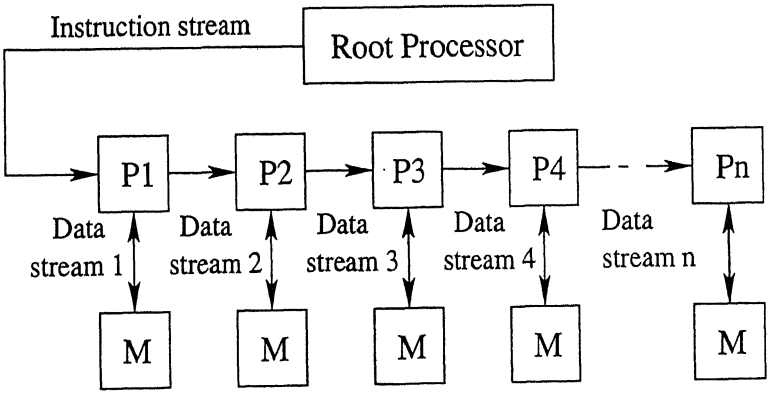


Figure 3.3: Schematic of a distributed memory SIMD computer.

Multiple-Instruction stream, Multiple Data stream (MIMD). This architecture may again be subdivided into shared memory systems and distributed systems. The MIMD shared memory network is illustrated in Figure 4. This implementation has the disadvantage that processors may have to queue to access the same area of memory.

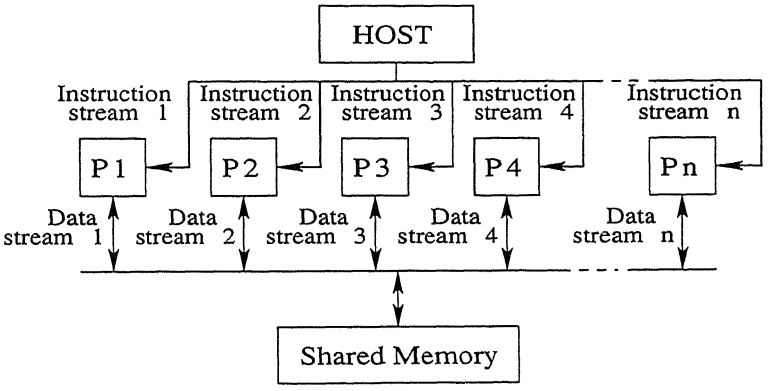


Figure 3.4: Schematic of a shared memory MIMD computer.

The distributed memory systems applies to systems such as transputer networks where each processor has its own local memory and communicates with other processors through channels, as shown in Figure 5.

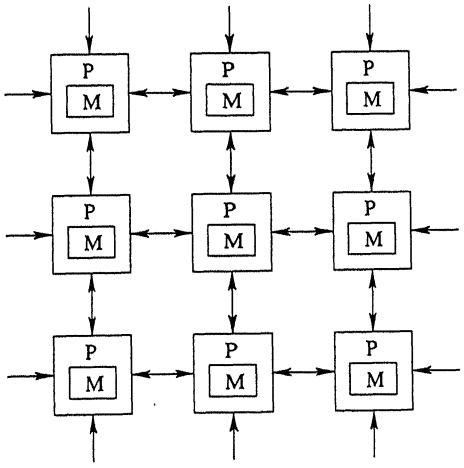


Figure 3.5: Schematic of a distributed memory MIMD computer.

For many problems programs in the individual processors of an MIMD system may be identical(or nearly so). Thus all the programs are carrying out the same operations on different set of data, just as SIMD machines would do. This

gives rise to the **Single-Program Multiple-Data(SPMD)** model of computation,also known as data parallel model.

Multiple-Instruction stream, Single Data stream (MISD). This architecture has yet to be implemented.

There is no obvious limitation to the number of processors in a MIMD architecture, but practical restrictions will be imposed by computational inefficiencies, communication requirements and hardware constraints.

As already indicated the computer architectures above may be further classified as:

- **Local or distributed memory systems:** where each processor carries its own own on-board memory which is not available to other processors in the network. Data transfer takes place through inter-processor communication.

- **Shared memory systems:** where a common memory area is made available to all the processors in the network and data transfer occurs through this shared memory. Performance tends to decrease rapidly with an increase in the number of processors.

The dominant computer architecture is currently MIMD with distributed memory. This implementation corresponds to a wide range of supercomputers, high performance multiple processor computers and networks of workstations.

3.2 Basic Concepts of Parallel Computing

Parallelism and Load Balancing: let us consider the problem of adding two n-vectors **a** and **b**.

$$a_i + b_i, \qquad i = 1, ..., n,$$

The additions are all independent and can be done in parallel. Thus, this problem has perfect mathematical parallelism. On the other hand, it may not have perfect parallelism on a parallel computer because of poor load balancing. By load balancing we mean the assignment of tasks to the processors of the system so as to keep each processor doing useful work as much as possible. Suppose for example that there are p=32 processors and n=100 as in equation above. Then processors can work in perfect parallelism on 96 additions but only 4 processors

will be active during the remaining 4 additions. Thus there is not perfect load balancing to match the perfect parallelism.

Related to load balancing is the idea of *granularity*. *Large-scale granularity* means large tasks that can be performed independently in parallel. An example is the solution of six different large systems of linear equations, whose solutions will be combined at a later stage of computation. *Small-scale granularity* means small tasks that can be performed in parallel; an example is the addition of two vectors where each task is the addition of two scalars.

Speed-up: ideally, we could solve a problem p times as fast on p processors as on a single processor. This ideal is rarely achieved; what is achieved is called the*speed-up of the parallel algorithm* defined by

$$S_p = \frac{\text{execution time for a single processor}}{\text{execution time using } p \text{ processors}}$$

The speed-up S_p is a measure how a given algorithm compares with itself on 1 and p processors. However the parallel algorithm may not be the best algorithm on a single processor. Hence a better measure is of what is gained by parallel computation is given by the alternative definition

$$S_p = \frac{\text{execution time on a single processor of fastest serial algorithm}}{\text{execution time of the parallel algorithm on } p \text{ processors}}$$

Though the above definition is more comprehensive. But the fastest possible algorithm may not be available to everybody. So, the previous definition is considered to be more practical and used mostly as the definition of speed-up.

Efficiency: closely related to speed up is efficiency, defined by

$$E_p = \frac{S_p}{p}$$

This is the measure of linearity of speed-up.

3.3 Available Algorithm

Among various popular parallelizable algorithm some are discussed below in brief.

3.3.1 Jacobi Iteration

To solve Laplace or Poisson equation which is frequently encountered in fluid mechanics and heat transfer problems, using Jacobi's method is the simplest iterative method. The solution is started with some initial guess. It is point-by-point method and within an iteration count *all* the values used are of previous iteration, instead of the latest updated and being used to update values (as in Gauss-Seidel method). For this reason its convergence rate is somewhat slow. But this highly parallelizable due to the fact that within an iteration count there is no need to communicate the updated values, as in Gauss-Seidel iteration. So for solving Laplace or Poisson equation in parallel, Jacobi's method is more attractive than Gauss-Seidel.

3.3.2 Domain Decomposition

The basic approach of domain decomposition is that a large domain is divided into many subdomains that are linked at the interfaces. The governing equations are solved independently in each subdomain with assumed interface conditions. A crucial factor in domain decomposition lies in its interface treatment. Among the various interface treatments available for domain decomposition, Uzawa's algorithm has been shown to be robust and parallelizable. It is analytically well-behaved and shows monotone convergence properties.

3.4 ANULIB Software

ANULIB software has been developed by BARC Mumbai, for the purpose of supporting several independent PC's/workstations in a parallel computing environment.It is necessary for such software a heirarchy and network of communications between the seperate processors that comprise the parallel environment.

The ANULIB programming follows the master-slave approach. Here one processor is the 'master' processor which gives order to other processors who are called slaves. The node on which user is working, is by default is the 'master' and other processors are slaves. The master and the slaves must have thier own seperate program. Like any other meassage passing software it is not a programming language. It is a library, available in both Fortran 77 and C programming

language. It consists of some functions.

Initialization and Termination Calls: To initialize any parallel programming environment, we should establish the communication channel between the chosen master and slaves. Corresponding calls are *m_init*() for the master, and *s_init*() for the slave program. The master has two arguments in the call giving the number of slaves and the slave's executable file name. Termination calls are *m_end*() and *s_end*() respectively, which are used when the computations in parallel is terminated.

Communication Calls: There are three types of communication calls for three different types of data, namely element, elements and data. An element is any valid Fortran data type. When just one element is sent and received between the processors *send_element*() and *receive_element*() calls are invoked. To send more than one such data types using a single call the corresponding send and receive calls are *send_elements*() and *receive_elements*(). In these two previous calls integer, real or any other combination can be sent and received simultaneously. Data in ANULIB is equivalent to arrays in Fortran or C. To send and receive arrays *send_data*() and *receive_data*() calls are used respectively. These two are more important than any other communication calls. Function calls for sending and receiving data are:

call send_data(data, data-size, destination, data-type)
call receive_data(data, data-size, source, size_check, data-type)

Arguments in the calls are self-explanatory. The *size_check* argument in the receive call is a safety parameter. It will check wheather the *data-size* (2nd argument of the receive call) written is same as the data received. Any integer is put here. Through out the text we have put it as "mptr", which is a null argument initially used.

Broadcast calls are used when same data there has to be sent to all the processors. But their use is always expensive over normal "sends".

Time calls: Two types of time measurement calls are used in ANULIB: *second()* will give the cpu-time and to measure the real-time *second1()* is used. Their use have been shown in the following section.

ANULIB Data Types: Data types in ANULIB look different from their Fortran equivalents. They are used only in communication calls. When sending an "integer" in the Fortran program, it is sent with data type "INT" in the communication call. "D_REAL" in ANULIB is equivalent to "double precision" in Fortran, etc.

3.4.1 More on Using ANULIB

The main thing about ANULIB is that simplicity is its beauty. It contains the least possible calls. When comparing with other message passing softwares it lacks some advanced and useful features. But there is always a way out. Some important issues are taken up and discussed below.

Passing of data: While sending data, they must be contiguous in memory. There is no problem when we send and receive a whole matrix like a(m,n) from the master (0) to slave 1. The corresponding send and receive calls

call send_data(a, m×n, 1, D_REAL)
call receive_data(a, m×n, 0, mptr, D_REAL)

will work fine. But the problem arises when we try to send a part of a matrix, as is often done in matrix-matrix multiplication computed parallely. While multiplying matrix a(m,n) with matrix b(n,q), copy of 'b' should be kept or sent to all the processors. The 'a' matrix will be divided between all the processors. Now, this matrix 'a' can be divided both horizontally (Figure 3.6) and vertically (Figure 3.7), but there is problem in sending a part when dividing horizontally while using Fortran as the programming language. Because the data is not contiguous in memory according to Fortran compiler's convention, which stores elements of 'a' matrix in column-major ordering. They are stored in the order a(1,1), a(2,1),.....a(m,1). Then the second column as a(1,2),.....a(m,2) and so on. So sending a horizontally cut matrix is problematic.

However sending a verticallly split matrix is simple. While sending such a matrix, the address of the first element of a division (contiguous in memory) is found and sent along with specifying the number of data to be sent. Thus when we have a b(25,25) matrix and five processors, for a vertically split matrix the

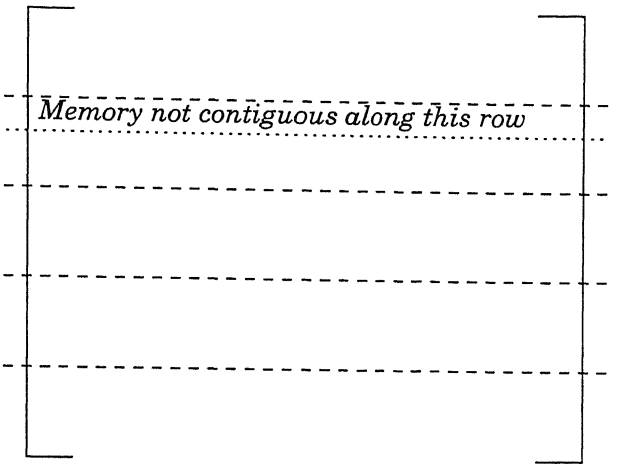


Figure 3.6: Schematic of a horizontally split matrix.

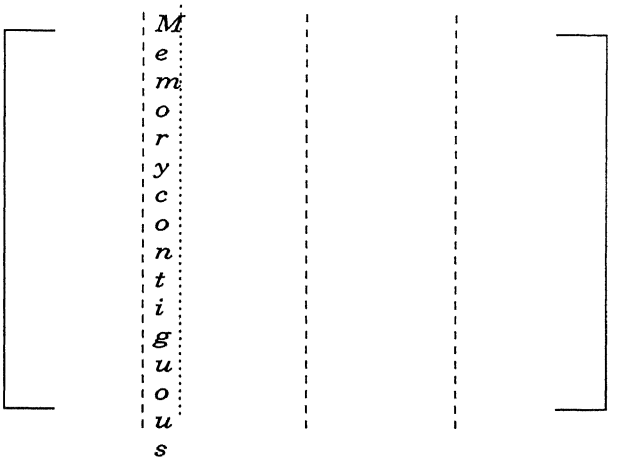


Figure 3.7: Schematic of a vertically split matrix.

master will keep the first 5 columns and sent the second set of 5 columns to the second processor with the corresponding call

call send_data(b(1,6), 125, 1, D_REAL)

where b(1,6) is the address of the first element of second part of the split matrix. The following number (125) in the call takes care of the number of data to be sent. So the elements in the 6th to 10th column will reach slave 1. Similarly other columns are sent to slaves 2, 3, 4.

Now the obvious question is, whether a horizontally split matrix part can be sent at all. It can be sent. In that case the data's memory has to be made contiguous first. A buffer memory is created which is contiguous in memory and will store the non-contiguous memories as its elements and that will be sent. The

process below explains it in program form.

```
dimension buffer(5,25)
do i=1,5
do j=1,25
buffer(i,j)=a(i,j)
end do
end do
```

Then the master sends it to slave 1 and the slave receives as follows

```
call send_data(buffer, 125, 1, D_REAL)
call receive_data(buffer, 125, 0, mptr, D_REAL)
```

The slave then puts it back to the desired location of matrix 'a'.

```
do i=1,5
do j=1,25
a(i,j)=buffer(i,j)
end do
end do
```

But creating buffer memory is always expensive. It will take some memory and while putting the values of the split 'a' matrix in buffer, the computer starts searching the whole 'a' matrix to get the required elements out of it and put those in the buffer. First it searches the first column then the second column and so on. This searching process is quite expensive when it has to be done for a large number of times, which is typical to unsteady flow/pseudo-transient CFD simulations.

Recording of Calculation and Communication Time : The computer code explains how to calculate both calculation time and communication time.

```
t1=second1()
do i=1,n
c(i)=a(i)+b(i)
end do
t2=second1()
```

```
t_cal=(t2-t1)
t1=second1()
call send_data(a, m×n, 1, D_REAL)
t2=second1()
t_comm=(t2-t1)
```

Here t_cal is the real calculation time and t_comm is the real communication time.

Inter Subroutine Communication between Master and Slaves: While writing a parallel program one may face the need of communication between a subroutine of a master to another subroutine in the slave or vice-versa. The subroutine in the master will include the file 'mincl.inc' and the subroutine in the slave will include 'sincl.inc' files. They essentially will establish communication channel between those subroutines. In a program there may be several such subroutines. Every time these files has to be included.

3.5 Application to Poisson Equation

Many CFD algorithms ultimately end up in solving a Poisson equation for pressure correction. And in our case the same type of equation is encountered. So solving a Poisson's equation using Parallel Processing technique was taken up as the first step toward parallelising the finite volume Navier Strokes solver.

3.5.1 Description of a Poisson's Equation and Its Finite Difference Discretization

A 2-D Poisson's Equation is shown below.

$$\frac{\partial}{\partial x}(h\frac{\partial p}{\partial x}) + \frac{\partial}{\partial x}(h\frac{\partial p}{\partial x}) = q$$

Where h and q are both constants or functions of independent variables. The equation is solved along-with suitable boundary conditions depending on the physical problem.

A finite difference discretization, 2nd order accurate in space will finally give a linear algebric equation like one shown below.

$$p(i,j) = a * p(i-1,j) + b * p(i+1,j) + c * p(i,j-1) + d * p(i,j+1) + e * q$$

Where a through e are constants and depends on the chosen grid. While solving the equation on a 2-D rectangular domain, an ideal mesh is shown below in Figure 3.8

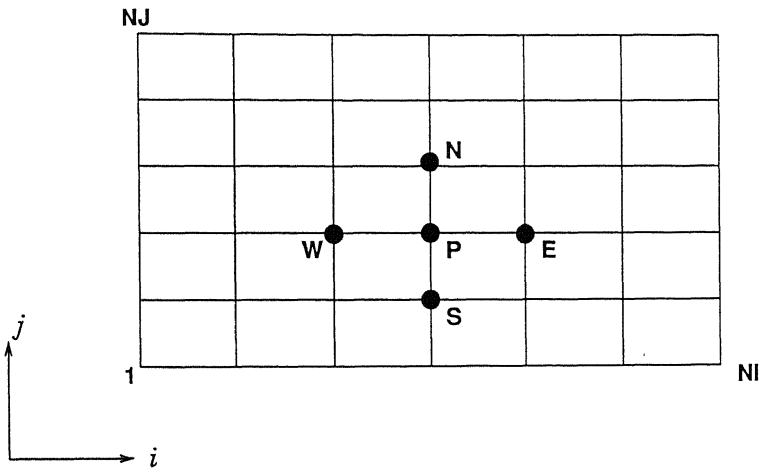


Figure 3.8: Schematic of the computational domain for the Poisson equation.

Keeping easy parallelizability of Jacobi's method in mind, the equation is solved by a point by point method with values of the four neighbours being the previous iteration values. The domain is split in some smaller sub-domains. Before moving further towards parallelisation take a look at the sequential algorithm.

3.5.2 Structure of the Sequential Code

1. Declaration of required variables, arrays and take input in form of parameters.
2. Grid lengths are calculated.
3. Initialization of pressure p and a dummy variable 'dumb' to store previous iteration value of p. Also put h and q according to their value.
4. Start the iteration count and set the error equal to zero.
5. Apply BC's.
6. Update p using Jacobi's method.
7. Define $error = \mathrm{mod}((p^{n+1} - p^n)/p^n)$
8. Check whether the biggest error is smaller than the convergence limit set earlier.

9. If error $\leq$ convergence limit, stop further iterations and store the value of pressure, else go to step 4 and continue.

3.5.3 Parallelization Strategy

In order to calculate the above algorithm in parallel we split the afore-mentioned physical domain into some smaller sub-domains and perform iteration in them simultaneously in different processors.

Domain Splitting Strategy: In any of the i or j directions we will split the domain into same-sized subdomains. The number of grid points in that direction should be chosen in such a way that it is evenly divisible by the number of processors. This takes care of load balancing during computation.

Now the obvious question is how to split, horizontally or vertically? It will depend on the direction of i and j in the finite difference mesh chosen. Before we discuss further how the direction of i and j is going to affect our splitting strategy, take a closer look, how the dependent variable p at a certain point is updated. It is updated through the previous iteration values of the variable p at its four neighboring (Figure 3.8) points values of p. But when P as shown in the figure is an interface point, how is it going to manage to get previous iteration values of the dependent variable from a neighbor? Because it is in some other slave/node.

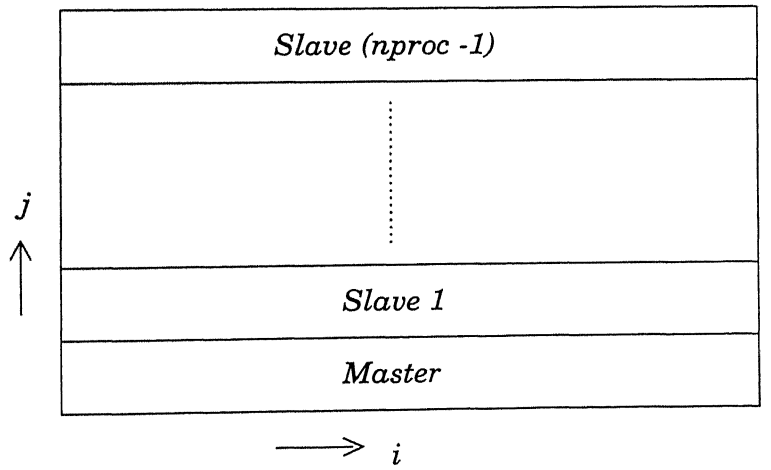


Figure 3.9: Schematic of a horizontally split domain.

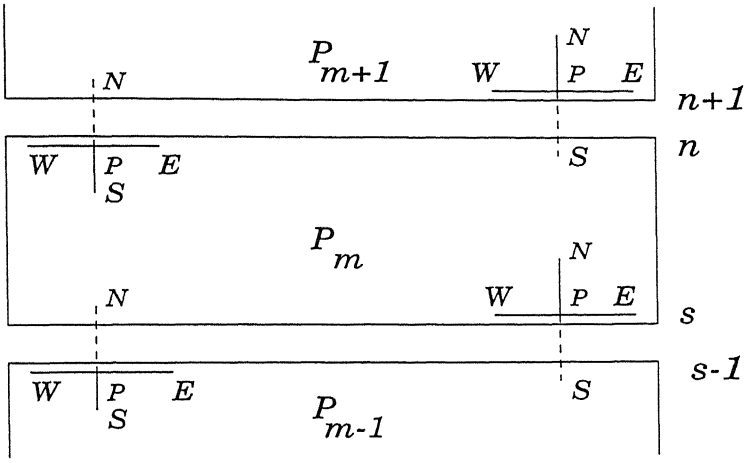


Figure 3.10: Schematic of data dependency among the processors.

Basically not a single point but the whole interfacial row points require the information of the interfacial row of its adjacent neighbour. In Figure 3.10, to update dependent variable at 'n'th row (of P_m th node) require data from '(n+1)'th row (of P_{m+1} th node). Similarly sth row (of P_m th node) require data from '(s-1)'th row (of P_{m-1} th node). In the same way to update p at '(n+1)'th row and '(s-1)'th row require previous iteration data from 'n'th row (of P_m th node) and 's'th row (of P_mth node) respectively.

Only nodes corresponding to extreme physical boundaries require information from one node only. Generally the master (process ID zero) node handles the lowest subdomain and the uppermost subdomain is handled by the last (process ID nproc-1) node. But any other combination is possible.

To handle these data-dependency at the interfacial rows, proper communication channel has to be established to send and receive ineterfacial row data. The data to be sent has to be contiguous in memory taking i as the horizontal axis and j as the vertical axis, data along each row is contiguous. So taking advantage of this contiguousness, the domain is split horizontally. It can also be splitted vertically but then interfacial columns to be sent are not contiguous in memory. A buffer has to be created as stated earlier, which will lead to a greater communication overhead.

If there are NI points in the i direction and NJ points in the j direction the width of each subdomain (divided horizontally) is defined as

$$nbnproc = NJ/nproc$$

Where *nproc* is the number of processors. The master will calculate from $j =$ *1,nbnproc*, (at $j = 1$ boundary condition is applied).
The calculation domain for each of the slaves is calculated as follows :

$$myid = get\_cpu\_id()$$
$$jinit = 1 + myid * nbnproc$$
$$jfinal = (myid + 1) * nbnproc$$

Where *myid* is processor ID, and slave executable file kept at slaves 1, 2, 3 will return *myid* 1,2,3 respectively, *jinit* is the initial j-index in the j-direction and *jfinal* is the final index. As *myid* is different for different slave so according to the process ID of the slaves *jinit* and *jfinal* will vary. So in the parallel version of the Poisson equation code, before proceeding to a new iteration required interface values of *dumb* are updated through proper communication. *dumb* is an array that stores the previous iteration values of p throughout the domain. Every slave starts computation with the same initial guess value of p. First iteration values are updated with this initial guess. Before 2nd and other consecutive iterations starts, interfacial values are exchanged between the processors. The part of the codes handling the communication part is shown below.

For the master the communication is as follows:

```
call send_data(dumb(1,nbnproc), NI, 1, D_REAL)
call receive_data(dumb(1,nbnproc+1), NI, 1, mptr, D_REAL)
```

In the slave program communication is taken care in the following way :

```
if (myid .eq. nproc-1) then
call send_data(dumb(1,jinit), NI, myid-1, D_REAL)
call receive_data(dumb(1,jinit-1), NI, myid-1, mptr, D_REAL)
else
call send_data(dumb(1,jfinal), NI, myid+1, D_REAL)
call receive_data(dumb(1,jinit-1), NI, myid-1, mptr, D_REAL)
call send_data(dumb(1,jinit), NI, myid-1, D_REAL)
call receive_data(dumb(1,jfinal+1), NI, myid+1, mptr, D_REAL)
endif
```

Here in the slave program the first block in the if loop handles communication for the last processor dealing the domain attached to the top boundary. Thus it handles one send and one receive in every iteration.

In the above send and receive calls, each process sends data to the process on top and then receives data from the process below it. The order is then reversed, and data is sent to the process below and received from the process above. This strategy of sending and receiving is simple and mostly applicable when send and receives are *blocking* in nature. In blocking kind of communication on send can only be completed when the corresponding receive call is invoked at the destination. If the receiver is not ready to receive it will wait still the receive call is invoked. So it must be ensured that the receiver processor reach that point of receiving when a send call is made from any of the processors. Another thing, when a processor is sending a data to another, at the same time it should not receive data from the same processor. There is always a chance of failure of such communication scheme, or otherwise may be a point of deadlock in the program.

Another important issue is taken up here. Let us consider the following code.
In master program :

call send_data(dumb(1,final), NI, 0, D_REAL)

In the slave program :

call receive_data(dumb(1,jinit-1), NI, 1, mptr, D_REAL)

In this case 'dumb' is being sent to slave 1 from the master (0). But it may happen that the process 1 is still doing some calculation and has not reached the point of receiving. So obvious question will arise what can process 0 do? The answer is process zero will stop calculation and wait until process 1 is ready to receive the message. This will have a detrimental effect on the speed-up. Here comes the ability the programmer to understand the inherent parallelism in the code and distribute it equally among all the processors and make a good synchronization among them.

Going back to Poisson's Equation we have said that before a new iteration starts the communications should be over and the interfacial data should be ready. This method of updating interfacial data before every iteration is called *synchronous* communication.

As the communication time among the processors is the biggest overhead in a data dependent algorithm, a potentially attractive alternative is to allow the processors to proceed *asynchronously.* That is instead of waiting for p^n iteration values at the interfaces , calculate p^{n+1} iteration values using the p^{n-1} iteration values at the interfaces. Even p^{n-2} or lower iteration values may be tried.

This new iterate is not the Jacobi iterate. However this may not be overly detrimental to the overall iteration and it may be cost effective in some cases to allow iteration to proceed asynchronously.

3.5.4 Convergence of Poisson's Algorithms

The solution is said to be convergent when

$$| p^{n+1} - p^n | \leq \epsilon$$

where ϵ is a predefined small number.

To calculate the above convergence parameters in the sequential codes has no complexity because information of p's in the whole domain is available. But in a parallel algorithm, no processor has the information of p's of other subdomains except its own. So to test convergence in the above fashion, information of ps of all the other processors needs to be sent to one common node. Then calculate the global convergence parameter and proceed accordingly. But this is extremely costly in terms of communicationand may be lead to very low speed-up or no speed-up at all.

What is done is that local convergence parameter (for the subdomain) is calculated and sent to a certain processor (master in our case) and determine the biggest among them. Then send it to all other processors and check whether it satisfies the convergence parameter. This way we can avoid burdensome communication to a good extent.

3.6 Parallelization of the 3-D Finite Volume Code

Many things encountered during the parallelization of the Poisson equation is faced again in parallellising the finite volume code. Issues exclusive to this specific problem is now taken-up one by one.

3.6.1 More on Memory Contiguousness

Previously we have discussed the contiguousness of data in an array in 2-D. Before we attack a 3-D problem let us discuss the contiguousness of 3-D array. The thumb rule considering Fortran compiler is that data is stored contiguously

with the outermost index varying first, then the next outermost index, etc. All the outer q indices can be varied keeping the innermost n-q indices fixed without sacrificing contiguousness in an n dimensional array.

In our code arrays are declared with 'i' being the innermost index, 'j' being the middle index and 'k' the outermost index. If in the geometry 'i' is the axial direction, 'j' is the theta direction and 'k' is the radial direction, then data on every slab with a fixed 'i' is contiguous. According to the direction of i, j, k chosen in geometry and position of those index in the array will decide the plane with contiguous memory.

3.6.2 Geometry Specific to our Problem

The geometry of the physical domain in the CVD problem is a pipe of circular cross section, with a length being double of the diameter. Inside the pipe the substrate block is kept at approximately one diameter distance. But that is not fixed and its position can be changed easily.

The program for the master and for the slaves is almost same but they simultaneously operate on different data, and these data is exclusive to some part of the physical domain. Only at interfaces communication is required. This model of parallelisation is called single program multiple data (SPMD). The SPMD model is adopted where there are low scale *granularity* in the program. Granularity of a program is, how its subroutines can run independently without data from another subroutine. For example if a program is having six algebric solvers then they can be run independently on six processors without any communication during their computation. They interact only after one step calculation is over and for next decision to come. This can be called as large scale granularity. In our program only one of the subroutines (pressure corrcetion) take the maximum time during execution.

3.6.3 The Best out of Three

The geometry has already been discussed. This geometry can be parted in three ways, axially, radially and sector wise. One with the minimum communication requirement will win the bid. Concerning the way the domain will be decomposed.

Axially Split Domain: The interface geometry is circular. Number of cells

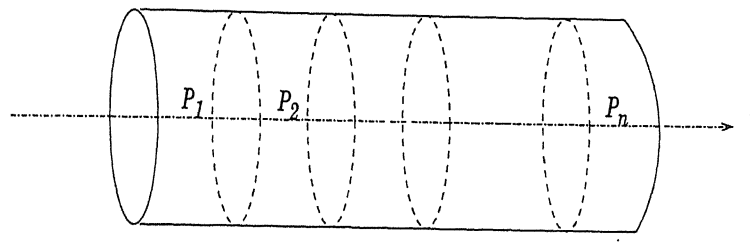


Figure 3.11: Schematic of axial splitting of the domain

forming the interface is NcellJ×NcellK. For every intermediate processor (2 interfaces) 2 pairs of send and receives (1 pair of send and 1 pair of receive) is required. For the two extreme sub-domains (one at inlet, other at exit) communication requirement is 1 pair of send-receive (one send and one receive).
So everytime an interfacial data communication is made, it will handle

$$\{2+2(\text{nproc-}2)\}\text{NcellJ}\times\text{NcellK}$$

number of data. Here *nproc* is the number of processor and NcellI, NcellJ and NcellK are number of cells respectively in i, j and k directions. Data is also contiguous on this face. When there is no communication between the first and last processor, the arrangement is called non-cyclic. So the processor topology here is non-cyclic.

Radially Split Domain: Interface geometry is cylindrical here. Number

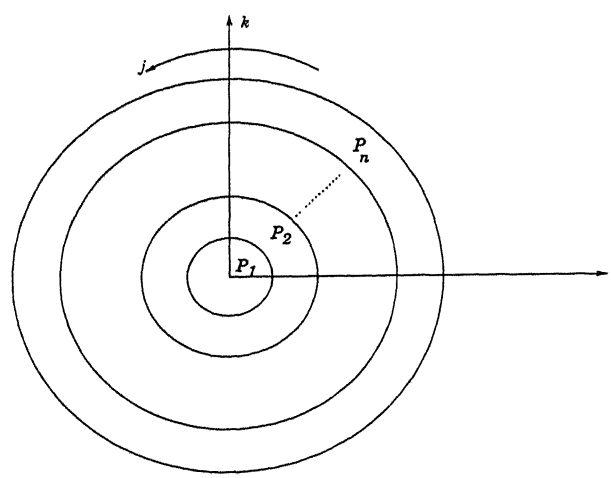


Figure 3.12: Schamatic of radial splitting of the domain.

of cells in this face is NcellI×NcellJ. Processor arrangement is non-cyclic. Total number of data to be passed is

$$\{2+2(\text{nproc-2})\}\text{NcellI}\times\text{NcellJ}$$

Data on this cylindrical face is non-contiguous. Using "buffer" is a must as interfacial data will be non-contiguous in memory.

Sectorwise Splitted Domain: Interface geometry is rectangular. Number

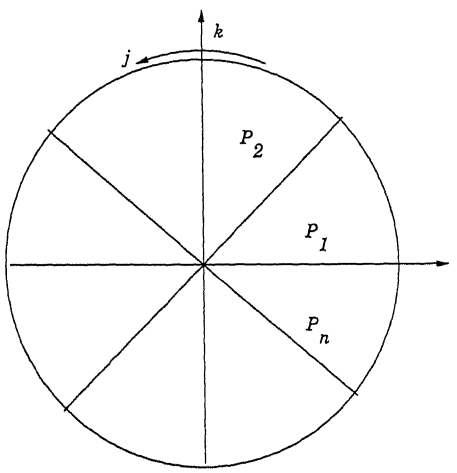


Figure 3.13: Schematic of sectorwise splitting of the domain.

of cells in this face NcellI×NcellK. Processor arrangement is cyclic. Total number of data to be passed is

$$2\times\text{nproc}\times\text{NcellI}\times\text{NcellJ}$$

Data on this face is non-contiguous. Using "buffer" is a must.

Considering the physical lengths and the flow physics number of grid points in the three different directions are such that

$$\text{NcellI} > \text{NcellK} > \text{NcellJ}$$

From the above discussion for each splitting, for a given number of processor and and above choice in number of grids, the communication requirement is lowest in the axially split domain. Moreover data on this face is contiguous. Another cause of its superiority is that, in this methodology the 3-D code is easily converted to 2-D code (2-D geometry is shown below) by merely fixing NcellJ and using symmetry in the j direction. 2 being layer of real cells and 1 and 3 are layers of fictitious cells.

A radially split 3-D parallel code can also be converted easily to 2-D parallel code but that splitting strategy is already discarded considering the communication cost.

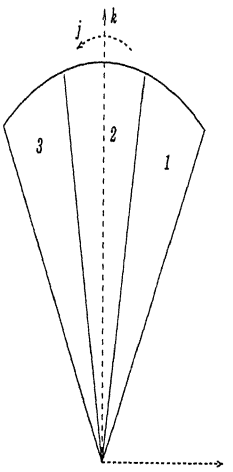


Figure 3.14: Schematic of the 2-D domain for the problem.

So the domain is split axially. Two fictitious cells are left. So $(NcellI - 2)$ number of cell is equally divided among the processors.

3.6.4 Parallelisation of the Predictor Step

Being an incomressible Navier-Stokes equation, it essentially handles convective and diffusive flux of three velocity components and gradiants of pressure. Following the predictor-corrector philosophy of the code, in a time step the predictor step predicts the velocity field and corrector corrects that field to enforce continuity. Predictor step always predicts on the basis of previous time step values. In this process of predicting velocities it calculates convective and diffusive fluxes of all the three velocities and normal derivatives of pressure on every cell face and for all the cells. This is for the full domain. Convective flux in the code is calculated using a hybrid scheme, combining central difference and a first order upwind scheme.

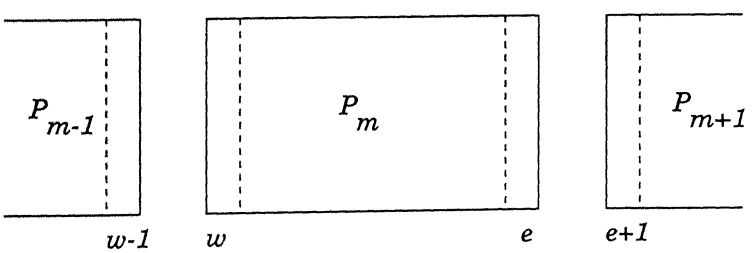


Figure 3.15: Schematic of data dependancy among the subdomains.

Following the schemes given in chapter 2: While calculating in parallel the

convective flux across the east face of the cells corresponding to 'e' location (of the P_m th processor) velocities and volumes of the 'e+1' location cells (of P_{m+1} th processor) is required. The reverse is also true. This shortage of data is fulfilled through proper interprocessor communication. Volume is calculated only once throughout the run of the code and every processor calculates the volume for all the cells of the full domain. So need for volume communication is eliminated and all the three velocity components of 'e+1' location cells are sent to processor P_m from P_{m+1} and then the process is reversed. For the convection scheme used here , data of one layer of cells from neighbouring processor is required. For other higher order schemes the communication requirement is higher and data of more than one layer of cells is required depending on the scheme used. These communication should be made before calculation of convective flux, so that in time of calculation the required data is available.

Diffusion has two components. One normal diffusion flux and the other is cross-derivative diffusion flux (see chapter 2). To calculate both the terms at an interface of any subdomain, information of velocities and pressure (pressure has only normal derivative component) from the neighbouring processor is essential. It should be taken care that information of all the four variables reach their proper location.

3.6.5 Parallellisation of Corrector Step

In the corrector step, we first calculate flux of momentum interpolated velocities at each face and for each cell (summing up all the six face quantities). In the predictor step, alongwith calculating the predicted velocities, momentum interpolated velocities are also calculated. To calculate flux across any face, velocity on that face is interpolated using the velocities at cell centre of the two cells adjacent to that face. For the 'e' location cells flux across the east face can only be calculated when information of momentum interpolated velocities at the cell centres of 'e+1' location cells are available.

Then the corrected net flux is nullified by the pressure corrections . So as to satisfy the continuity condition. So calculation of pressure gradiants is the next in the line. Its treatment is same as that of the diffusion flux of velocities, made earlier in the predictor step. So the communication requirement is similar. In

the effort to make the net flux in the each cell of the domain equal to zero, the pressure correction term is continually updated for each cell using the residual flux in that cell. This updation of the pressure correction term at every iteration must be reported for interfacial cells, to the corresponding neighbour, at the end of that iteration and prior to the next iteration. It will continue until the convergence in terms of flux has reached in a time step.

Convergence of Flux in a Time Step The residual flux is calculated for each cell and squared. Then it is added with other cell values and done for the whole sub-domain. This value for every processor is communicated to the master. The master adds them all, divides it by the number of cells in the whole domain. Then square root of this quantity is found and is communicated back to the slaves. If it is lower than the set convergence value, velocities and pressure is updated, otherwise the iterations continue.

Time Step Calculation in Parallel Time step is also calculated partly in parallel. With their velocities and grid dimensions every subdomain calculates its own time step combining both CFL and grid Fourier number conditions. Then sent to master to determine the smallest one and this time step value is informed to all processors. This way the calculation in every processor advance same amount in time.

In someways these calculations of time step and flux residual globally in the master are points of bottleneck in the code. But they are unavoidable. One thing that must be ensured, that every processor should reach these points in same point of time. That is taken care by proper load balancing.

Output files are updated parallely and this updation is made after interval of a prescribed number of time steps. When steady state results are the ultimate aim, they can be updated after reaching the steady state. It has been observed that they take a lot of time to update.

The way we have calculated the flux residual, time step and steadiness parameter in time, makes no difference between the parallel code and sequential code. This makes it easier to compare how the same algorithm behaves in sequential as well as in parallel.

Chapter 4

Results and Discussion

4.1 Overview

In this chapter the results of the Poisson equation are shown and some insight regarding the behaviour of the algorithm for various set of grids and processors is discussed for modes of communication namely *synchronous* and *asynchronous*. Then the CVD code is taken-up and its success both in terms of accuracy of predicting the physical flow and in terms of speed up and efficiency is discussed.

4.1.1 Poisson Equation

The Poisson equation taken up is a very general equation and was solved for three different cases depending on the values of h and q. The physical domain chosen is shown below.

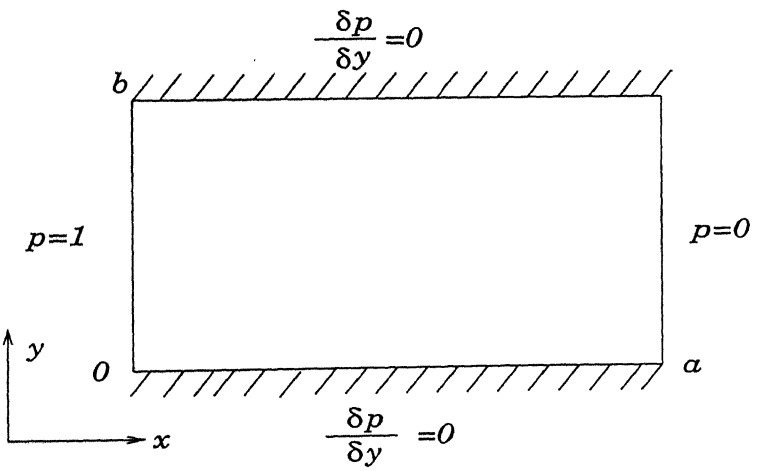


Figure 4.1: Physical domain of the Poisson problem.

Case 1:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0;$$

Here in this case $h = 1$ and $q(x, y) = 0$ .This is a *Laplace* equation.

Case 2:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = \sin \frac{\pi x}{a} sin \frac{\pi y}{b};$$

Where $h = 1$ and $q(x, y) = \sin \frac{\pi x}{a} sin \frac{\pi y}{b}$.

Case 3:

$$\frac{\partial}{\partial x}(h\frac{\partial p}{\partial x}) + \frac{\partial}{\partial y}(h\frac{\partial p}{\partial y}) = \sin \frac{\pi x}{a} sin \frac{\pi y}{b}$$

Here $h = \sin \frac{\pi x}{a} sin \frac{\pi y}{b} + 0.1$ and $q(x, y) = \sin \frac{\pi x}{a} sin \frac{\pi y}{b}$

All the three cases were solved on a 2-D square domain (a = b) with the following boundary conditions at its four boundaries.

(i) At $x = 0$; $p = 1$.
(ii) At $x =$ a; $p = 0$.
(iii) At $y = 0$; $\frac{\partial p}{\partial y} = 0$.
(iv) At $y =$ b; $\frac{\partial p}{\partial y} = 0$.

The solution algorithm have been discussed earlier in chapter 3. It was solved using Jacobi iteration.

Length in both the directions were take as unity i.e. $a = 1$ and $b = 1$. Number of grid points chosen were 80×80 and 120×120. For 80×80 grids the code was tested on 2, 4, 5 and 8 processors. For 120×120 number of processors chosen were 2, 3, 4, 5, 6, 8. For both the variation of grids and corresponding choice of processors, the distribution of load is equal as the number of grid points are evenly divisible in each case. All the three cases were tested in both synchronous and asynchronous mode of communication. In the asynchronous mode communication was made to occur in every two iteration. It is needless to mention that all the above cases were run on a single processor also. Variation of p along the axial

direction was measured. In all the above cases time recorded was real time and measured using the *second1()* function call of the **ANULIB** library as *second()* gives only CPU-time, and not communication time. Both the communication time as well as calculation time was recorded in using the real time call. It makes their comparison easier.

4.1.2 Time and Accuracy Comparison.

The time comparisons for cases 1, 2, 3 on various number of processors with the corresponding speed-up and efficiency for synchronous and asynchronous modes of communication are shown in Tables 1-12. The results achieved are shown against against the single processor results in 4.2 - 4.7.

Table 1: Time records in **Case 1**, with synchronous communication on 80 × 80 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	5519	6.210	-	-	-	-
2	5517	4.117	85.96	14.04	1.508	75.40
4	5517	3.531	45.80	54.20	1.758	43.95
5	5517	3.261	38.68	61.32	1.904	38.08
8	5517	2.808	24.86	75.14	2.211	27.64

Table 2: Time records in **Case 1**, with asynchronous communication on 80 × 80 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	5519	6.210	-	-	-	-
2	5595	3.668	79.02	20.89	1.693	84.65
4	5621	2.609	63.17	36.83	2.380	59.49
5	5633	2.320	54.71	45.29	2.676	53.52
8	5673	1.857	38.49	61.51	3.354	41.92

Table 3: Time records in **Case 1**, with synchronous communication on 120×120 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	9045	24.582	-	-	-	-
2	9040	17.633	84.64	15.36	1.394	69.7
3	9040	12.789	67.74	32.26	1.922	64.07
4	9040	10.660	56.80	43.20	2.306	57.65
5	9040	9.250	51.01	48.99	2.657	53.1
6	9040	8.460	45.97	54.30	2.896	48.27
8	9040	7.410	41.31	58.69	3.317	41.47

Table 4: Time records in **Case 1**, with asynchronous communication on 120×120 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	9045	24.582	-	-	-	-
2	9172	15.930	91.19	8.81	1.543	77.15
3	9181	10.762	77.73	22.27	2.284	76.13
4	9197	8.507	71.42	28.58	2.868	71.70
5	9207	7.320	68.73	31.27	3.358	67.16
6	9221	6.430	60.63	39.37	3.823	63.72
8	9247	5.402	57.19	42.81	4.550	56.87

Table 5: Time records in **Case 2**, with synchronous communication on 80×80 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	5376	5.879	-	-	-	-
2	5388	4.117	73.34	26.66	1.428	71.40
4	5391	3.652	38.10	61.90	1.610	40.25
5	5287	3.359	34.30	65.70	1.750	35.00
8	5368	2.769	27.64	72.36	2.123	26.54

Table 6: Time records in **Case 2**, with asynchronous communication on 80×80 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	5376	5.879	-	-	-	-
2	5397	3.723	83.10	16.90	1.579	78.97
4	5455	2.688	57.25	42.75	2.204	55.09
5	5473	2.328	51.75	48.25	2.526	50.52
8	5505	1.859	19.95	80.05	3.163	39.54

Table 7: Time records in **Case 2**, with synchronous communication on 120×120 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	8725	22.471	-	-	-	-
2	8745	16.669	82.92	17.08	1.348	67.41
3	8753	12.613	73.33	26.67	1.781	59.38
4	8751	10.410	57.41	42.59	2.158	53.96
5	8742	9.168	48.91	51.09	2.451	49.02
6	8729	8.293	47.85	52.15	2.709	45.16
8	8703	7.238	41.55	48.45	3.104	38.81

Table 8: Time records in **Case 2**, with asynchronous communication on 120×120 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	8725	22.471	-	-	-	-
2	8753	15.120	92.43	7.57	1.486	74.31
3	8761	10.301	76.71	23.29	2.181	72.71
4	8807	8.400	72.15	27.85	2.675	66.87
5	8835	7.121	64.89	35.11	3.155	63.11
6	8851	6.281	62.37	37.63	3.577	59.62
8	8865	5.273	54.66	45.34	4.262	53.27

Table 9: Time records in **Case 3**, with synchronous communication on 80×80 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	6835	14.125	-	-	-	-
2	6883	8.687	87.05	12.95	1.626	81.30
4	6917	6.094	56.41	43.59	2.318	57.95
5	6913	5.375	33.72	66.28	2.628	52.25
8	6899	4.844	33.54	66.46	2.916	36.45

Table 10: Time records in **Case 3**, with asynchronous communication on 80×80 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	6835	14.125	-	-	-	-
2	9172	8.031	90.66	9.37	1.758	87.93
4	9197	5.031	70.80	29.20	2.807	70.18
5	9207	4.281	56.93	43.07	3.300	66.00
8	9247	3.594	40.86	59.14	3.930	49.13

Table 11: Time records in **Case 3**, with synchronous communication on 120×120 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	9759	48.406	-	-	-	-
2	9853	28.687	87.14	12.86	1.687	84.37
3	9917	20.530	73.51	26.49	2.358	78.60
4	9936	17.156	68.85	31.15	2.821	70.54
5	9929	14.406	63.55	36.45	3.360	67.20
6	9918	13.625	56.19	43.81	3.553	59.21
8	9901	12.1875	56.41	43.59	3.972	49.65

Table 12: Time records in **Case 3**, with asynchronous communication on 120×120 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	9759	48.406	-	-	-	-
2	9985	28.280	89.17	10.83	1.712	85.58
3	10103	18.468	86.12	13.87	2.621	87.37
4	10121	15.000	79.79	20.21	3.227	80.67
5	10153	12.530	77.31	22.69	3.863	77.26
6	10145	11.718	69.59	30.41	4.131	68.85
8	10147	9.844	66.03	33.97	4.917	61.46

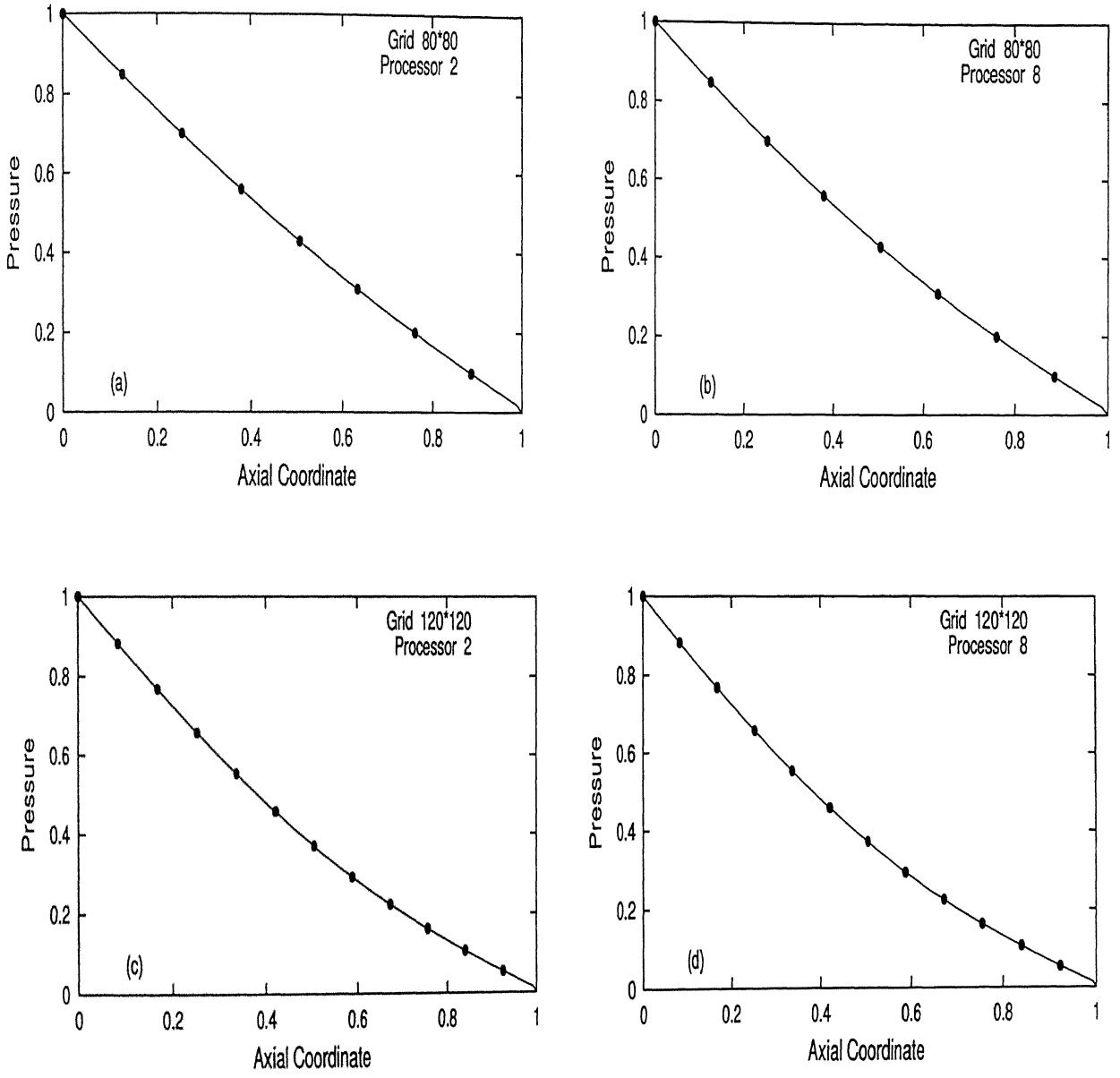


Figure 4.2: (a),(b),(c),(d) Comparison of parallely computed axial pressure variation (full curve) with result obtained from 1 processor (shaded circle).

Case 1 with Asynchronous Communication

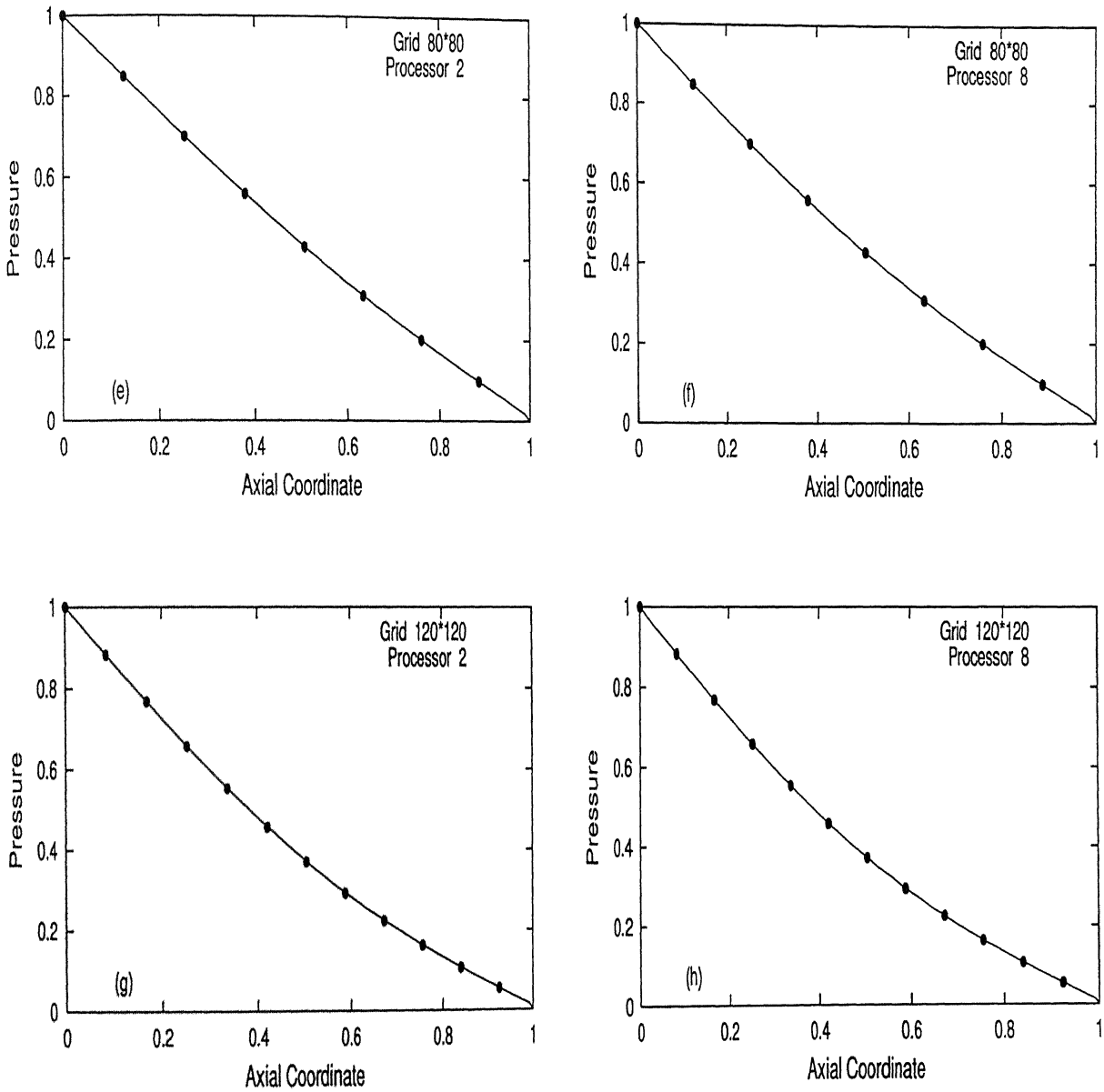


Figure 4.3: (e),(f),(g),(h) Comparison of parallely computed axial pressure variation (full curve) with result obtained from 1 processor (shaded circle).

Case 2 with Synchronous Communication

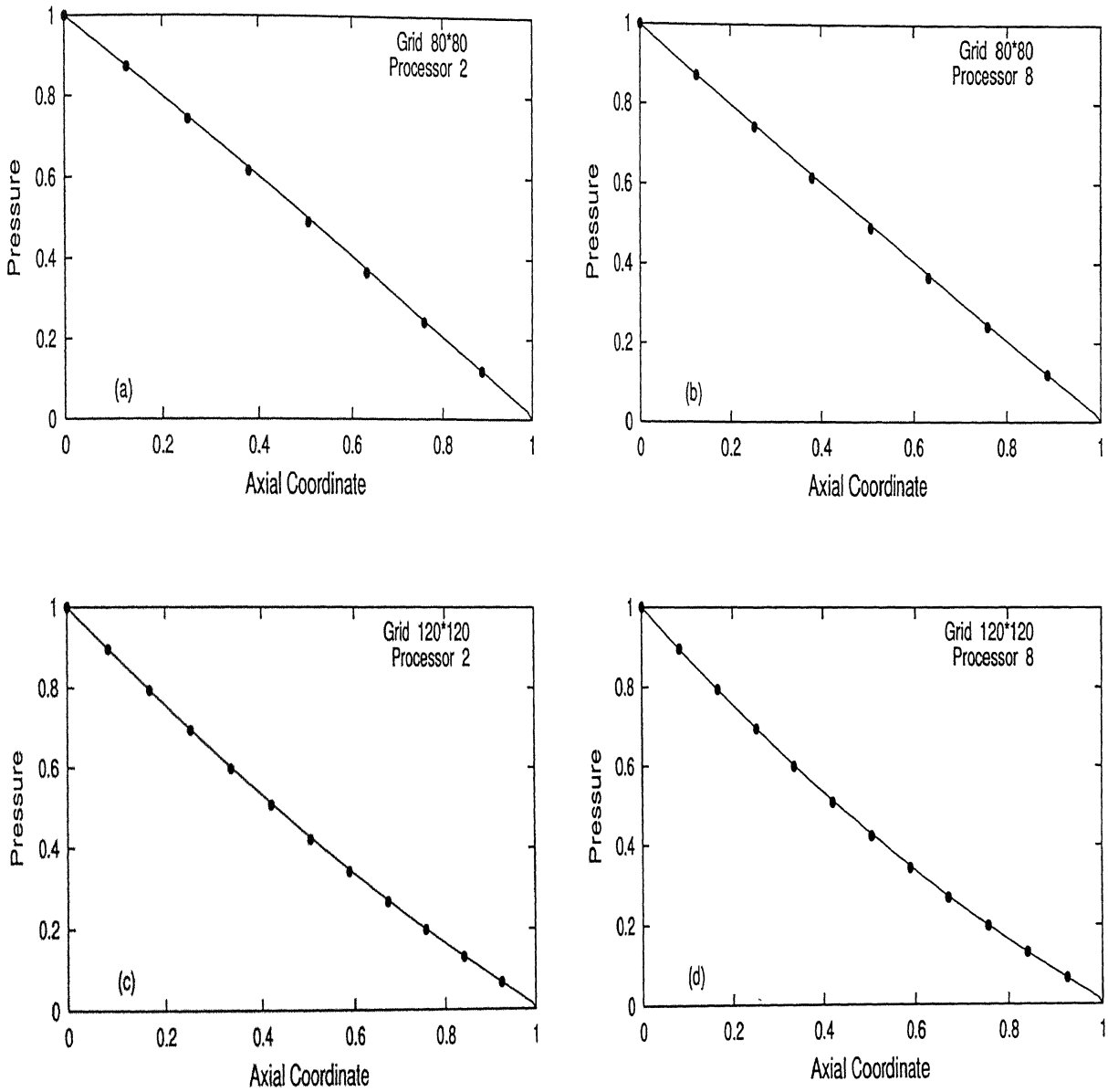


Figure 4.4: (a),(b),(c),(d) Comparison of parallely computed axial pressure variation (full curve) with result obtained from 1 processor (shaded circle).

Case 2 with Asynchronous Communication

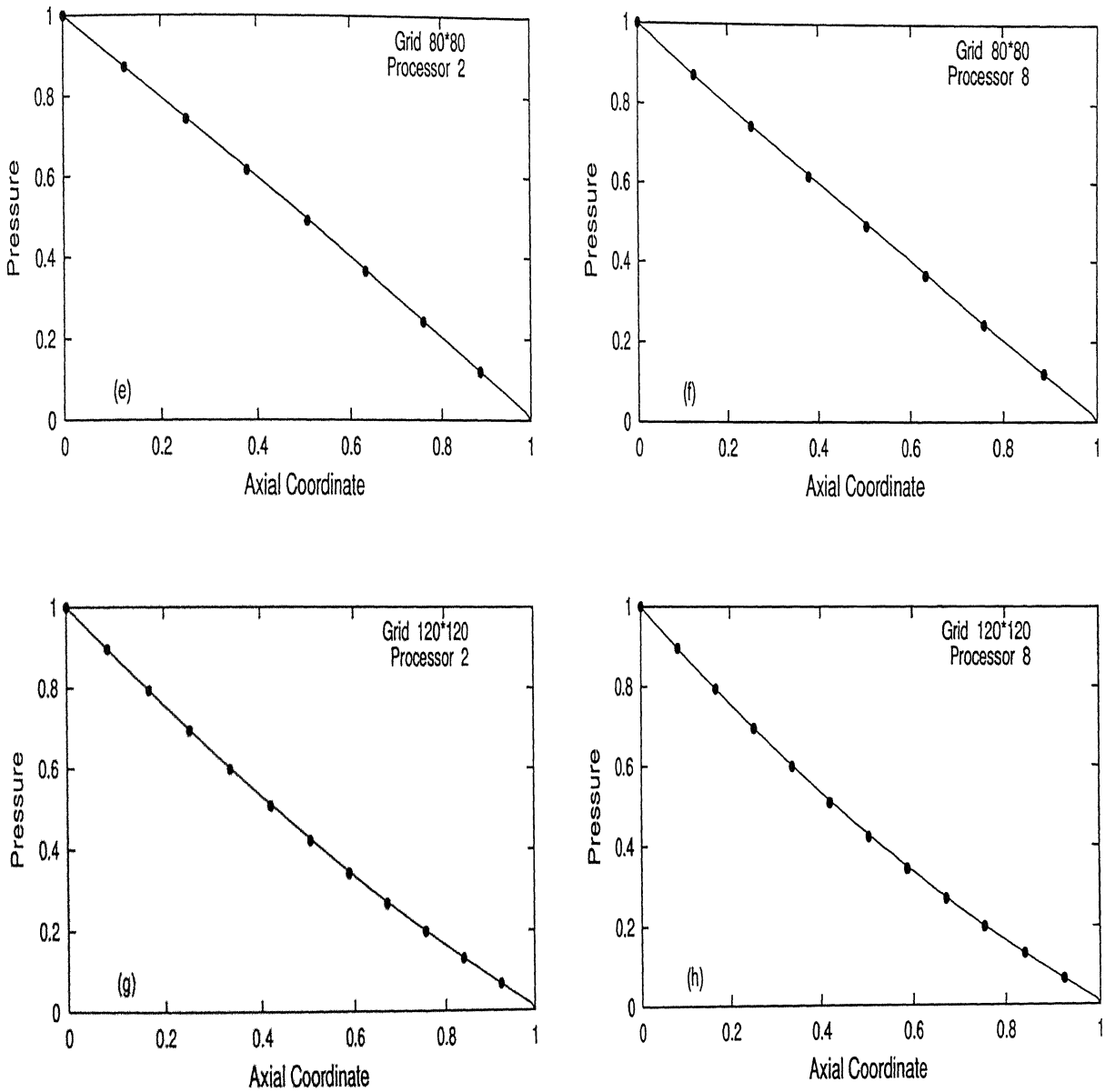


Figure 4.5: (e),(f),(g),(h) Comparison of parallely computed axial pressure variation (full curve) with result obtained from 1 processor (shaded circle).

Case 3 with Synchronous Communication

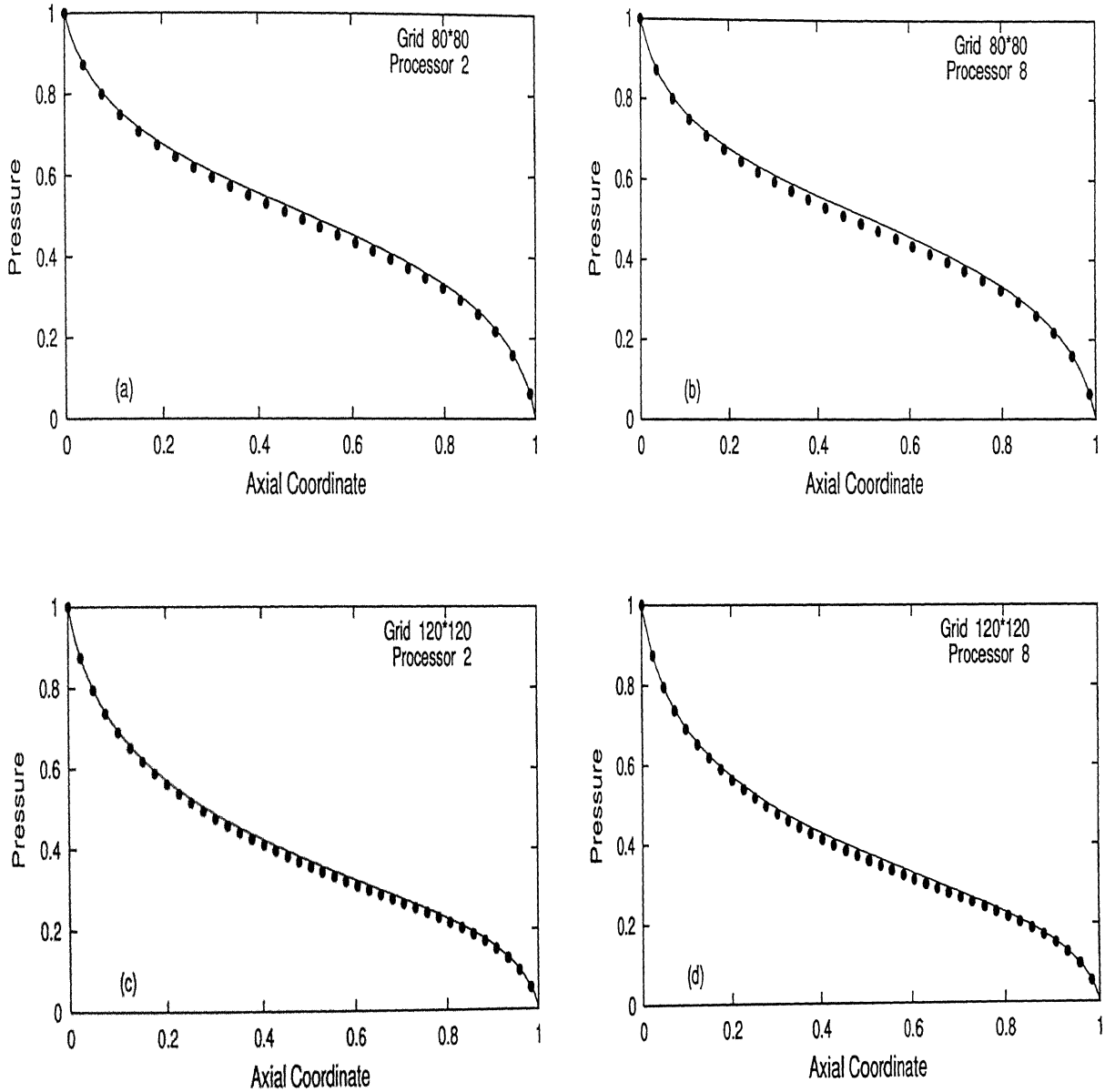


Figure 4.6: (a),(b),(c),(d) Comparison of parallely computed axial pressure variation (full curve) with result obtained from 1 processor (shaded circle).

Case 3 with Asynchronous Communication

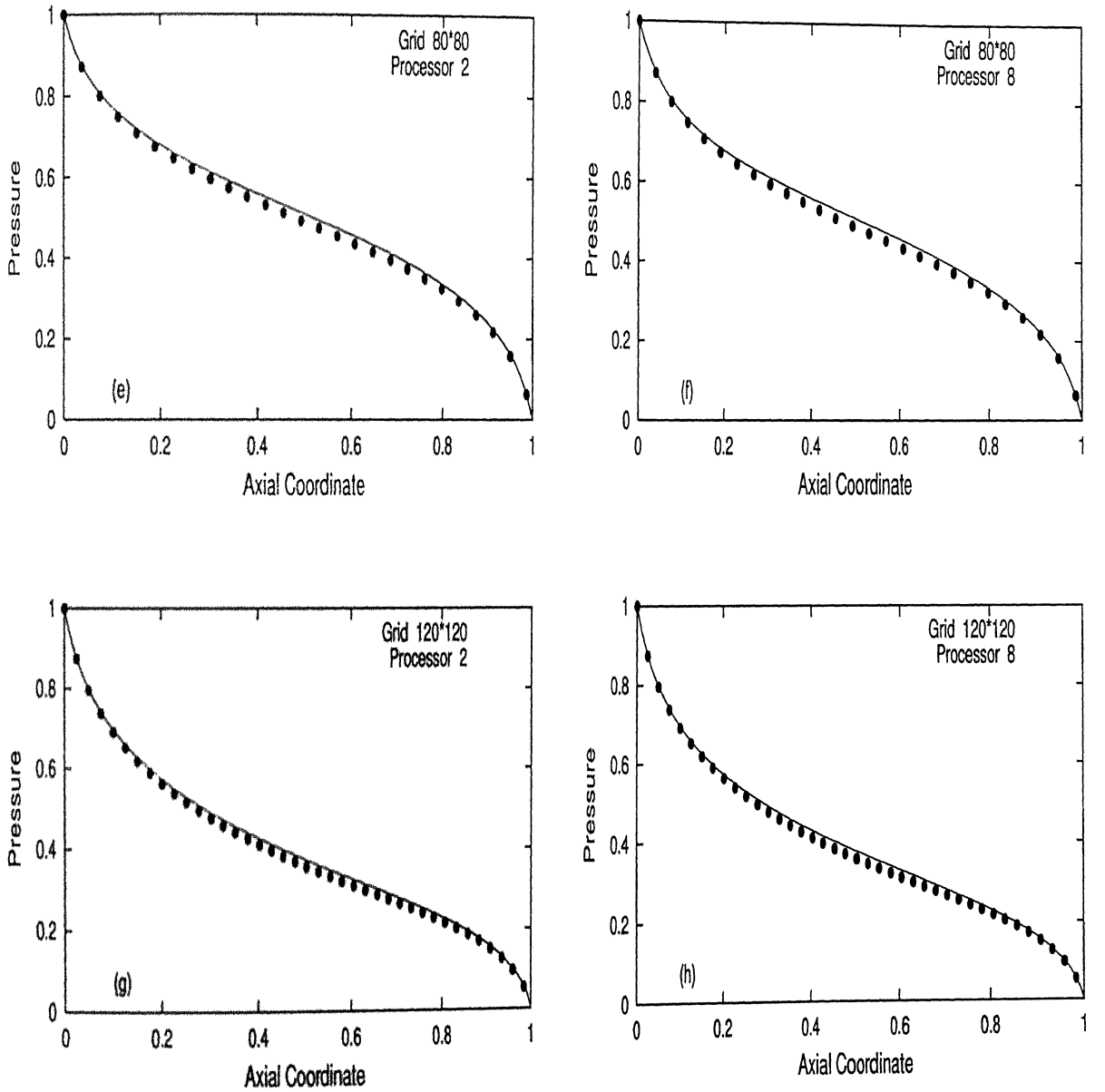


Figure 4.7: (e),(f),(g),(h) Comparison of parallely computed axial pressure variation (full curve) with result obtained from 1 processor (shaded circle).

Some trends are evident from the time-record Tables. As the number of processor is increased the speed-up certainly incresses, but not linearly; indeed the efficiency(= speed-up/number of processors) monotonically decreases with the increasing number of processors. Further asynchronous communication always gives better speed-up than synchronous communication for the same problem. Finally, less the computations involved in the problem, the less is the benefit of parallelisation. For example the speed-up for 8 processors, say, increases as the problem becomes more computer intensive from case 1 to case 3.

The decrease in efficiency with increasing number of processors, is easily explained. It is due to the problem being a data-dependent one thus requiring communication. Each time a communication call is invoked, calculation stops. This is the basic drawback of *blocking* send-receives. With grid points remaining the same, with increasing number of processors, more communication is needed while the calculation domain for each processor gets smaller. Both of this facts have a negative effect on the speed up. With calculation domain getting smaller calculation-time per iteration decreases, while communication increases due to an increase in number of interfaces. the same. In this context we define a relevant parameter, *volume-to-surface* ratio of a subdomain. It is the ratio of number of interior grid points for which no data communication is required, to the number of grid points which depends on data from other processors. In 2-D with increasing grid points the numerator of the above term increases quadratically and denominator linearly, thus a relative relative decrease in communication time.

Our results are explained by the above phenomena. As we shift form 80×80 grid points to 120×120, with domain length remaining the same in both the x-direction and y-direction, with same number of processors both speed up as well as efficiency increases. So one thing can be concluded, besides being better suited for highly time consuming codes, parallel programming practice is also better for bigger problems.

Between synchronous and asynchronous communication modes, the later one is clearly preferable in terms of speed-up and efficiency. although the number of iterations taken to reach the same convergence is often higher than in the synchronous mode. To consider the number of iterations, as a performance parameter is not wise where the sole aim is reduction in time. Results obtained from both synchronous and asynchronous modes have good match with the sequential result

in all the three cases. The gain from asynchronous communication, increases with the increasing number of processor. So employing asynchronous communication with more number of processor is much more profitable.

To demonstrate this as during our discussion we have opnied that parallel processing is for large problem sizes. Running problem of any small size in parallel is not wise. The three cases we have discussed so far was tested on 320×320 grid points. For 80×80 and 120×120 grid points, decrease in efficiency as we go from 2 to 8 processors is very sharp. With 2 processors in all the three cases and both in synchronous and asynchronous mode efficiency was as high as 70% or more. But with 8 processors it fell to around 40-50% and in some cases to even less than 30%. However, all the three cases showed better performance on 320×320 grid points. Even with 8 processors efficiency was never below 50% and change in efficiency from 2 to 8 processors is much gradual than those with the other two grids. See tables 13-18 in the following pages.

Table 13: Time records in **Case 1**, with synchronous communication on 320×320 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	20680	457.937	-	-	-	-
2	20630	318.310	94.45	5.55	1.438	71.93
4	20630	182.875	83.12	16.87	2.504	62.60
8	20630	107.750	67.24	32.76	4.250	53.12

Table 14: Time records in **Case 1**, with asynchronous communication on 320×320 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	20680	457.937	-	-	-	-
2	21177	321.765	95.86	4.14	1.423	71.16
4	21171	184.406	83.64	16.36	2.483	62.08
8	21171	100.218	71.26	28.74	4.569	57.12

Table 15: Time records in **Case 2**, with synchronous communication on 320×320 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	18136	417.937	-	-	-	-
2	18139	283.125	92.82	7.18	1.476	73.81
4	18109	165.047	80.29	19.71	2.532	63.30
8	17605	94.203	63.16	36.84	4.436	55.45

Table 16: Time records in **Case 2**, with asynchronous communication on 320×320 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	18136	417.937	-	-	-	-
2	18135	275.687	94.35	5.65	1.516	75.80
4	18101	158.015	84.10	15.90	2.644	66.12
8	18075	86.468	71.83	28.17	4.833	60.42

Table 17: Time records in **Case 3**, with synchronous communication on 320×320 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	13894	522.25	-	-	-	-
2	13860	292.937	97.32	2.68	1.783	89.15
4	13762	155.875	90.18	9.82	3.350	83.76
8	13501	85.906	78.12	21.88	6.079	75.99

Table 18: Time records in **Case 3**, with asynchronous communication on 320×320 grid points.

Processor	Iteration	Real Time	% of Calculation	% of Communication	Speed Up	Efficiency
1	13894	522.25	-	-	-	-
2	13877	289.562	98.29	1.71	1.803	90.18
4	14001	154.344	94.72	5.28	3.384	84.59
8	13879	84.000	81.53	19.47	6.217	77.77

We also observe from these tables that the efficiency gain, using asynchronous over synchronous communication for large number of processor is not that evident in the case of 320×320 grid points. This is because, for large number of grid points the calculation-to-communication time for each subdomain is much higher than for smaller subdomains in case of grid points such as 80×80 and 120×120. So any reduction in communication time is hardly felt in speed-up and efficiency.

4.2 Results for CVD Reactor

In this section we describe results obtained for various simulations on CVD reactor. The various cases are listed below.

1. 2-D straight block on two different grid sizes, each on two different Reynolds number of 50 and 100.

2. 3-D straight block, with Reynolds number of 50 and 100 on a single grid size.

3. 2-D curved block on the same grid size as of in 2-D straight block case with same two Reynolds number.

All the Reynolds number are based on the central jet velocity and the diameter of the reactor chamber.

The 3-D solution large computational time. It was shown by Viswadeep (2001) that flow in the 3-D reactor is essentially axi-symmetric. So later on, the code was converted to 2-D as in figure 4.8, just taking only one layer of real cells. The middle sector is the real computational domain and two boundary sectors are fictitious layer of cells taken to apply boundary condition. The angle subtended by each of these sectors at the centre of the pipe is taken 10 degree. Both central jet and peripheral jet is injected through the middle cell.

One approximation of this 2-D reactor in comparison with the 3-D is that, the four discrete peripheral cells turn to a ring-like continuous jet. But the time reduction was the main factor to shift this geometry. For the same Reynolds number, the numerical value of any physical quantity (such as skin friction coefficient) obtained from 2-D run and 3-D run may not match quantitatively but their qualitative trend is same. We will see this in coming sections.

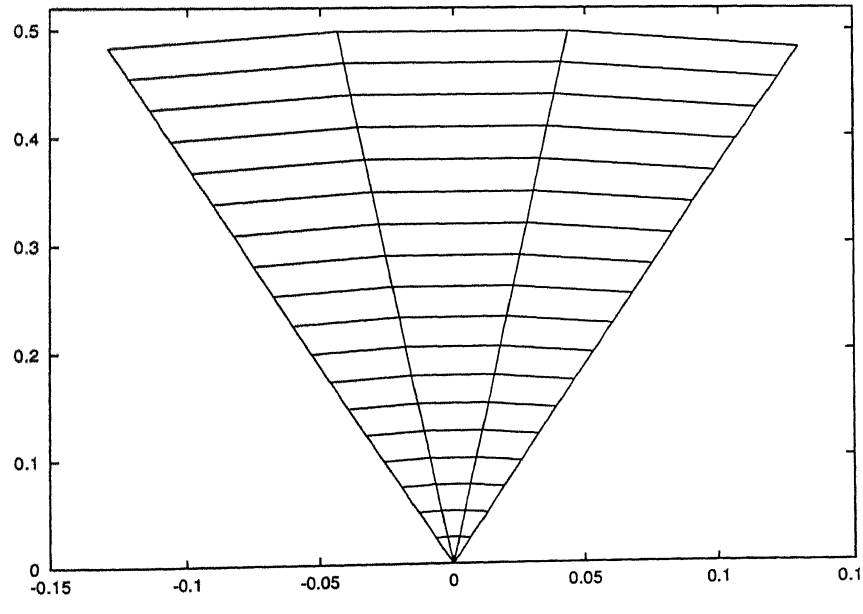


Figure 4.8: Schematic of 2-D grid arrangement.

To validate of the results obtained by parallel computation, we have matched the computed skin friction coefficient with the corresponding sequential result. Velocity vector plots are also used to compare sequentially and parallelly computed solutions. Both the skin friction coefficient and vector plots were taken on

the plane containing a pair of peripheral jets. Vector plots are shown only for the upper half section of the pipe.

Skin friction factor is defined as below:

$$C_f = \frac{\mu(\frac{\partial u}{\partial y})_{\text{wall}}}{\frac{1}{2}\rho u_{\text{avg}}^2} \tag{4.1}$$

Here $(\frac{\partial u}{\partial y})_{\text{wall}}$ is the velocity gradiant at the wall. u_{avg} is the average velocity in the flow domain. ρ is the density of the fluid.

The equation was non-dimensionalized. The non-dimensional form of the equation is

$$C_f^* = \frac{2}{\text{Re}} \frac{(\frac{\partial u^*}{\partial y^*})_{\text{wall}}}{u_{\text{avg}}^{*2}} \tag{4.2}$$

Here C_f^* is non-dimensional skin friction coefficient, $u^* = \frac{u}{u_c}$ and $y^* = \frac{y}{D}$. u_c being the central jet velocity and D being the reaction chamber diameter. In all our plots skin friction factor shown is the non-dimensionalised one.

The 2-D straight block CVD reactor: The CVD reactor configuration was tested on two different grid size of $53 \times 4 \times 21$ (coarse grid) and $108 \times 4 \times 31$ (fine grid). The numbers represent the grid points in the axial, angular and in radial directions, respectively. The grids were generated by a orthogonal grid generation method.

In all the straight block cases velocity ratio of the central jet to the peripheral jet was taken as 5 and peripheral jet was injected at an angle of 30 degree inwards. Central jet diameter was taken one fourth of the radius of the reactor chamber. Ratio of substrate diameter to the reactor chamber diameter is called blockage ratio. For all the straight bolck cases it was taken approximatly equal to 0.7. The block is taken to be placed at an approximately one diameter distance away from the reactor entrance.

For the coarse mesh, the peripheral jet was taken to be placed between 12 to 16 cells in the radial direction. For fine mesh the numbers are 18 and 24 respectively.

The 2-D curved block CVD reactor: The CVD reactor was tested with same two grid sizes. The grids were generated by a orthogonal grid generation method. The same block position, blockage ratio and position of central and

peripheral jets was used as for the straihgt 2-D block. For the curved block configuration peripheral jet was injected parallely with the central jet. The grid shown below is the coarse mesh.

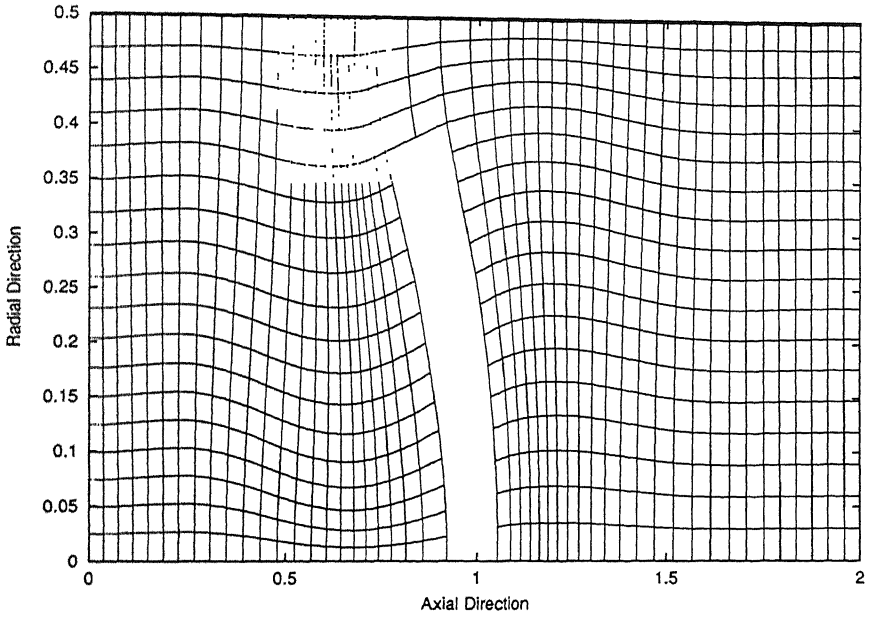


Figure 4.9: Schematic of a grid used in 2-D curved block computation.

The actual 3-D straight block reactor: The reactor was simulated on a grid size of $66 \times 19 \times 31$ for the chosen Reyolds numbers of 50 and 100. The number of grid points in the angular direction is chosen in such a manner, so that number of cells in the angular direction is evenly divisible by the number of peripheral jets. It was tested on 3 and 7 processors as well as on a single computer. Block position, blockage ratio and other parameters remains the same. The peripheral jet was injected at an angle of 30 degree inwards.

Before we discuss the parallel performance parameters such as speed-up and efficiency, we must make sure that results obtained from the parallel computation are identical with the sequential computation. If not identical, they must be within reasonable limits.

In all our cases the match is well within resonable limits. In some cases it is exact. In most of the fine mesh cases the match is perfect, and for coarse mesh cases the difference is within reasonable limits. This small variation is due to the insufficient number of grids to capture the physical phenomena. The skin friction and velocity vector plots shown in each case, for parallel and sequential results

are very nearly identical. See figures 4.10 to 4.33.

The Reynolds number rouhgly signifies inertia force over viscous force. So Re = 50 flow is much viscous than the Re = 100 case. That is also reflected in our plots. In all our cases wallshear is higher for Re = 50 than in Re = 100.

4.2.1 Speed-up and Efficiency:

All the cases considered, follow similar trends in speed-up and efficiency. So, instead of discussing them individually we prefer to discuss them as a whole. Any special observation in any of the cases will be discussed seperately.

The speed-up and accordingly efficiency is comparatively lower for all the cases when run on coarse mesh (see tables 19 - 22). For the coarse mesh cases the number of grids points per subdomain is smaller thus communication to calculation time ratio is higher. *Volume-to-surface* ratio for coarse mesh is 9 and for fine mesh even with 7 processor it is 14 (*volume-to -surface* ratio is the ratio of number of grid points whose calculation is independent of communication, to the number of points whose calculation is dependent on data from other processor). The smaller the ratio, the more the number of points dependent on data from other processor.

Results were also taken for 1000 time steps (T1000), from 101th time step to 1100th time step. In this range of time steps, number of iteration taken in the corrector step per time step is only one both in parallel and serial runs. As the total number of iterations taken to reach the steady state is different for parallel and serial one, T1000 is a proper measure of speed-up. However, the speed-up and efficiency data given in the tables are based on overall time to reach steady state, as we are interested in reducing the overall time.

In the coarse grid case of Re = 50, speed up based on T1000 is 61.78% (calculated from table 19), whereas the overall efficiency is higher (65.41%) than this. This is due the fact that overall time step taken for steady state solution have reduced in parallel computation, thus a significant gain in terms of reduction of time.

For the fine grid (Re 50) also the same phenomena is observed. For 3, 5 and 7 processors efficiencies based on $T1000$ are 86.6, 73.15 and 69.27% (calculated from table 20) respectively. Looking at the number of time steps to steady state shown

in the tables, it is clear that in computation in parallel has led to accelerated convergence towards steady state.

Another observation is that on same number of grid points, T1000 both in sequential and in parallel is almost equal for both the Reynolds numbers of 50 and 100. However, the overall time taken for Re = 50 it is higher than for Re = 100 is due to the time step size for Re = 50 is less than for Re = 100.

In percentage terms, for 3, 5 and 7 (case 2-D straight block) processors % of communication for Re 50 case are 10.54, 15.79 and 24.26 respectively. The corresponding values for Re 100 are 10.94, 15.20 and 25.1. (The deviation for 5 processor is due to that one of our computers is performing much slow compared to other computers. This phenomena was noticed at much later stage of my work, when most of the final runs were complete.)

Another observation is that in the 3-D straigh block case we get more efficiency compared to 2-D cases using same number of processors. This is easily understood. We have already defined *volume-to-surface* ratio. For a 2-D fine grid the above mentioned ratio while running on 3 processor is 10.26 $((33 \times 28)/(30 \times 3))$ and for the 3-D domain is 16.32 $((19 \times 16 \times 29)/(18 \times 30))$. For 7 processors the ratios in 2-D and 3-D are 4.04 $((13 \times 28)/(30 \times 3))$ and 5.807 $((7 \times 16 \times 28)/(18 \times 30)$. So in both with 3 and 7 processor we get a higher efficiency in the case of 3-D, as the volume to surface ratio (and thus the computation to communication ratio) is higher.

Time Records for Both the Grids and Both the Reynolds Number in 2-D Straight Block Case

Table 19: Time records for 2-D straight block, grid points $53 \times 4 \times 21$ on 1 and 5 processors for Re = 50.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	37193	2660.07	-	57.77	-	-
5	35309	813.40	247.21	18.7	3.27	65.41

Table 20: Time records for 2-D straight block, grid points $108 \times 4 \times 31$ on 1, 3, 5 and 7 processors for Re = 50.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	83656	19502.13	-	189.75	-	-
3	79066	7170.58	756.32	73.04	2.719	90.66
5	79066	4879.04	770.70	51.88	3.997	79.94
7	79066	3914.62	953.82	39.13	4.982	71.18

Table 21: Time records for 2-D straight block, grid points $53 \times 4 \times 21$ on 1 and 5 processors for Re = 100.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	28628	2239.97	-	57.81	-	-
5	26506	686.94	213.11	19.35	3.261	65.21

Table 22: Time records for 2-D straight block, grid points $108 \times 4 \times 31$ on 1, 3, 5 and 7 processors for Re = 100.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	61517	15753.95	-	187.68	-	-
3	58330	5689.70	622.62	72.69	2.769	92.29
5	58330	4062.83	617.71	51.59	3.877	77.55
7	58330	3103.75	779.06	42.98	5.076	72.51

Time Records for Both the Grids and Both the Reynolds Number in 3-D Straight Block Case

Table 23: Time records for 3-D straight block, grid points $66 \times 19 \times 21$ on 1, 3 and 7 processors, Re = 50.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	22611	66363.89	-	1558.42	-	-
3	22335	22734.83	1568.15	535.53	2.919	97.30
7	22335	11757.06	2206.13	291.00	5.640	80.64

Table 24: Time records for 3-D straight block, grid points $66 \times 19 \times 21$ on 1, 3 and 7 processors for Re = 100.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	19090	57158.17	-	1400.04	-	-
3	18783	22349.70	1469.41	540.08	2.557	85.29
7	18783	10862.17	1912.31	287.04	5.262	75.17

Time Records for Both the Grids and Both the Reynolds Number in 2-D Curved Block Case

Table 25: Time records for 2-D curved block, grid points $53 \times 4 \times 21$ on 1 and 5 processors for Re = 50.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	154229	9409.95	-	58.33	-	-
5	142846	2666.26	637.41	17.20	3.529	70.58

Table 26: Time records for 2-D curved block, grid points $108 \times 4 \times 31$ on 1, 3, 5 and 7 processors for Re = 50.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	275729	54700.72	-	179.19	-	-
3	256636	18930.04	1640.38	66.78	2.889	96.32
5	256337	13201.47	2768.86	49.00	4.143	82.87
7	256636	10965.68	4059.65	41.98	4.988	71.26

Table 27: Time records for 2-D curved block, grid points $53 \times 4 \times 21$ on 1 and 5 processors for Re = 100.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	123300	7736.28	-	58.11	-	-
5	108095	2229.93	1068.74	18.99	3.470	69.40

Table 28: Time records for

2-D curved block, grid points $108 \times 4 \times 31$ on 1, 3, 5 and 7 processors for Re = 100.

Processor	Step	Real Time	Communication Time	T1000	Speed Up	Efficiency
1	215834	43677.67	-	174.95	-	-
3	200249	15275.74	1272.20	65.59	2.859	95.31
5	199781	9645.85	1661.28	42.20	4.528	90.56
7	200363	9096.33	3295.08	40.95	4.801	68.59

Skin Friction Coefficient Comparison in 2-D Straight Block, Re = 50.

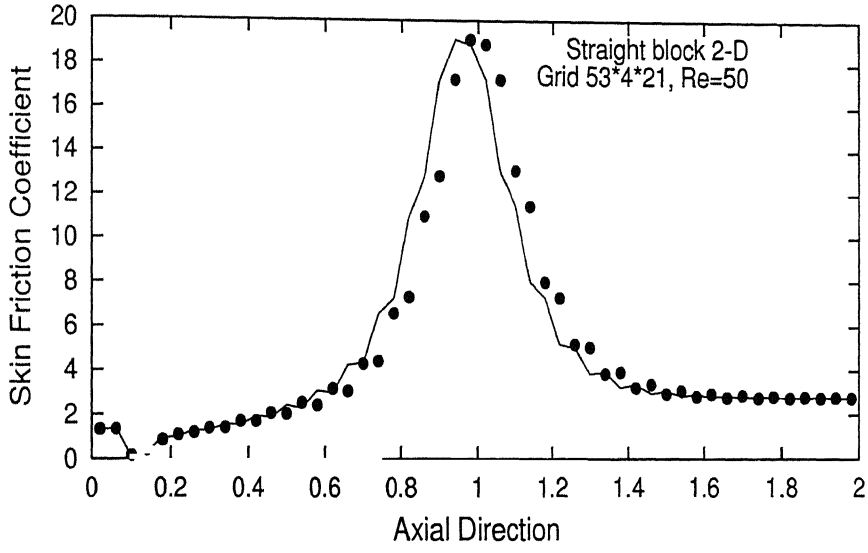


Figure 4.10: Comparison of parallely (5 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

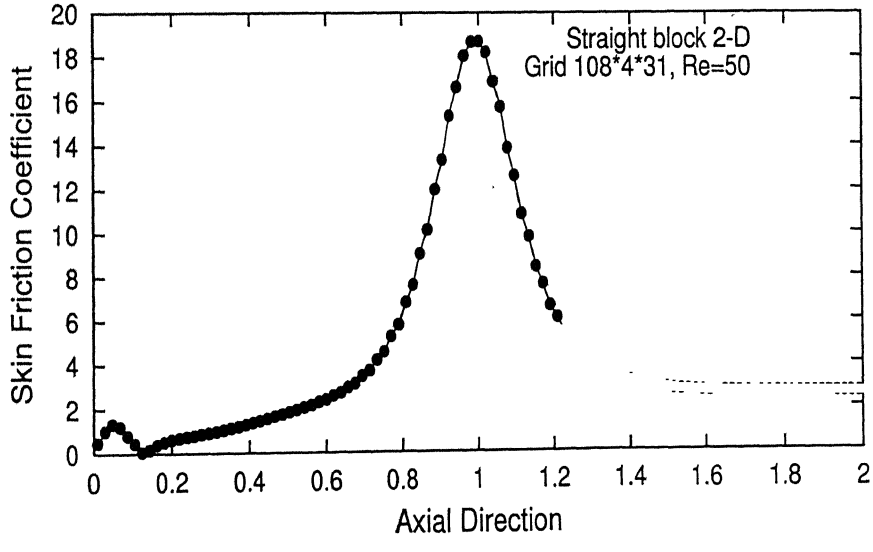


Figure 4.11: Comparison parallely (3 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

Velocity Vector Plot for 2-D Straight Block, Re = 50.

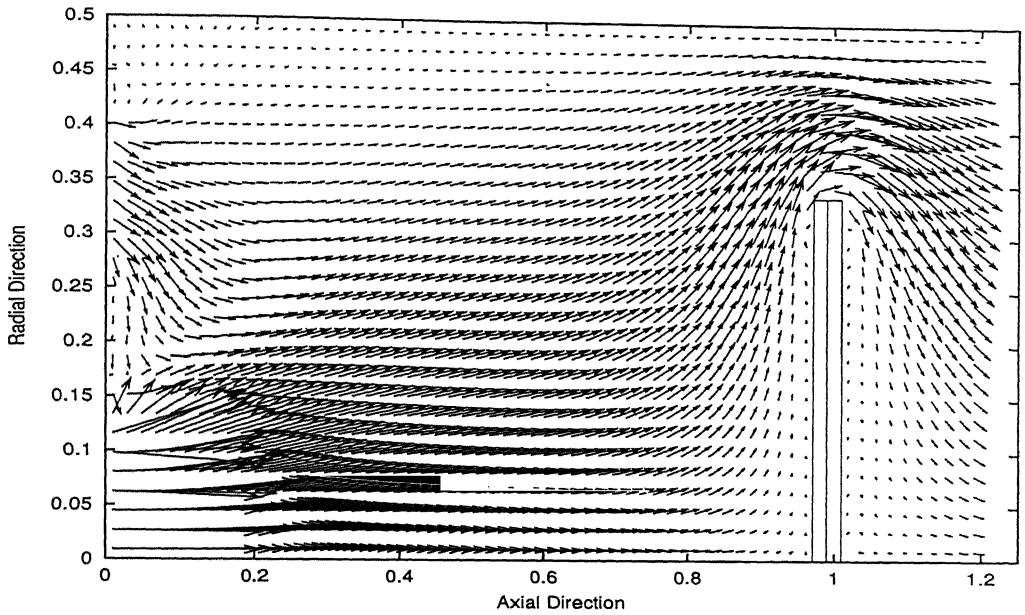


Figure 4.12: Parallely computed (5 node) vector plot on fine grid.

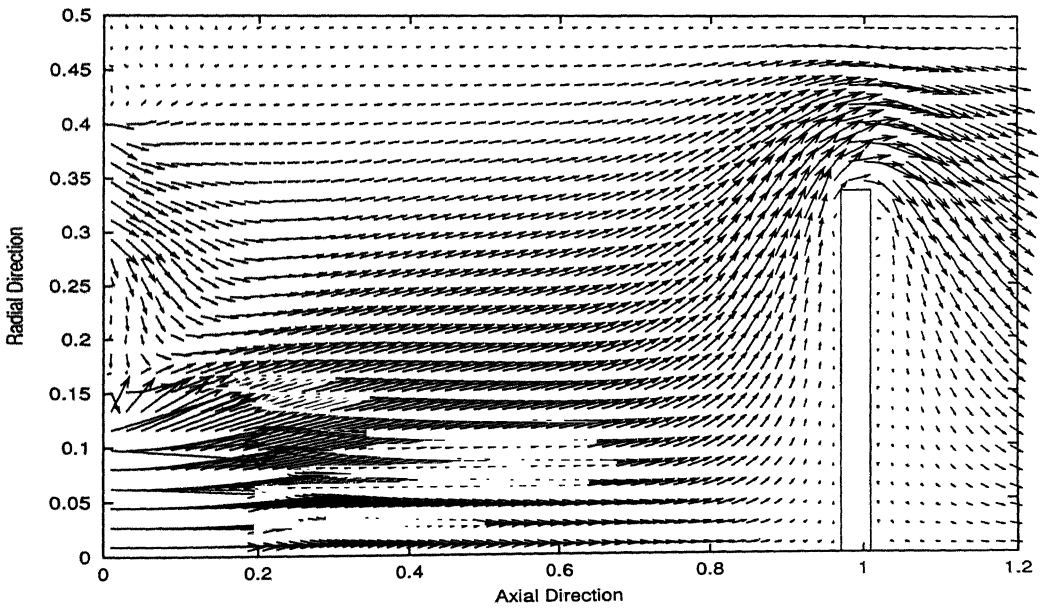


Figure 4.13: Sequentially computed vector plot on fine grid.

Skin Friction Coefficient Comparison in 2-D Straight Block, Re = 100.

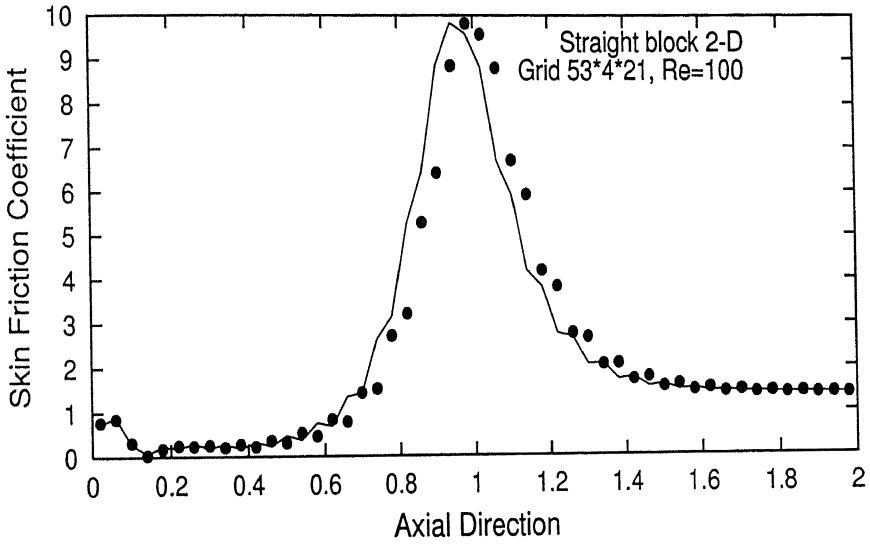


Figure 4.14: Comparison of parallely (5 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

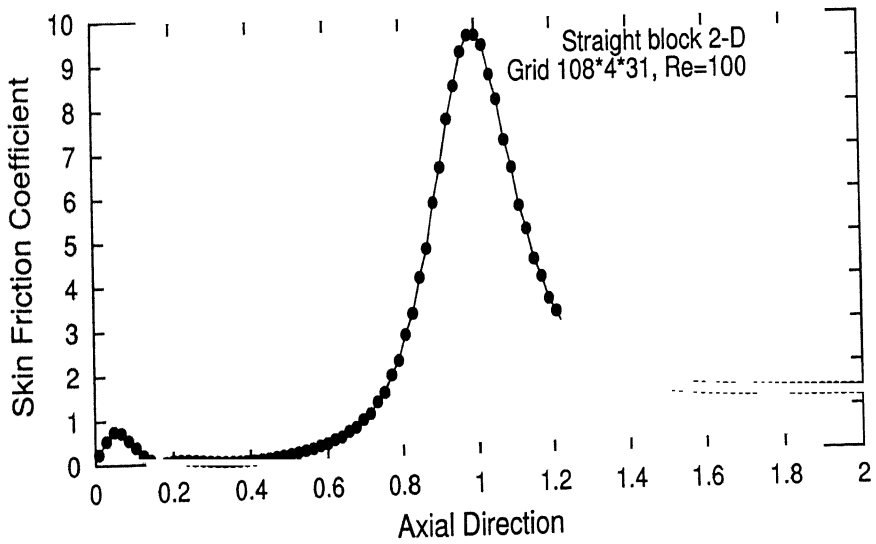


Figure 4.15: Comparison of Skin friction parallely (3 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

Velocity Vector Plot for 2-D Straight Block, Re = 100.

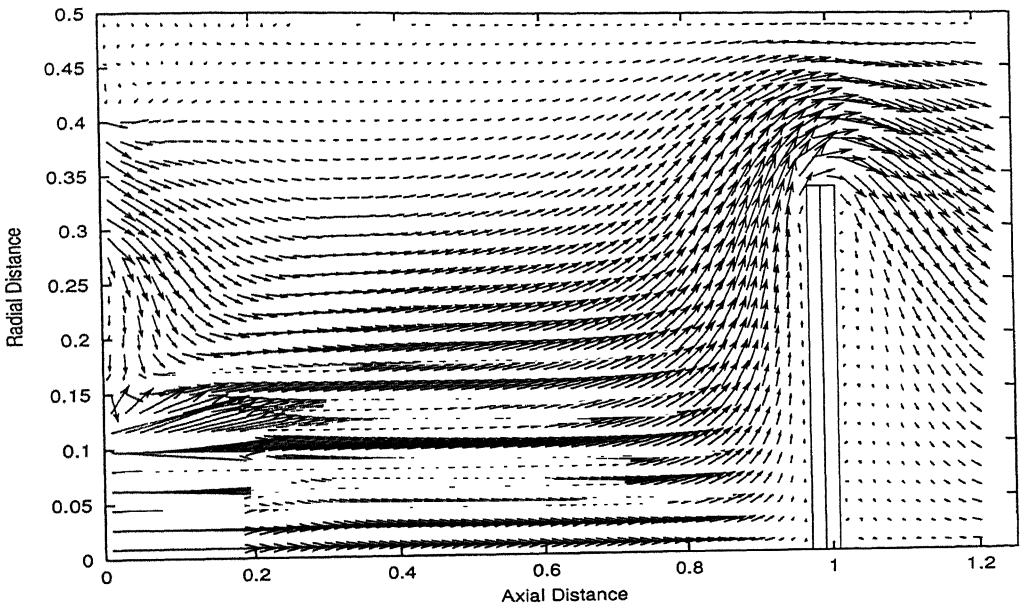


Figure 4.16: Parallelly computed (5 node) vector plot in fine grid.

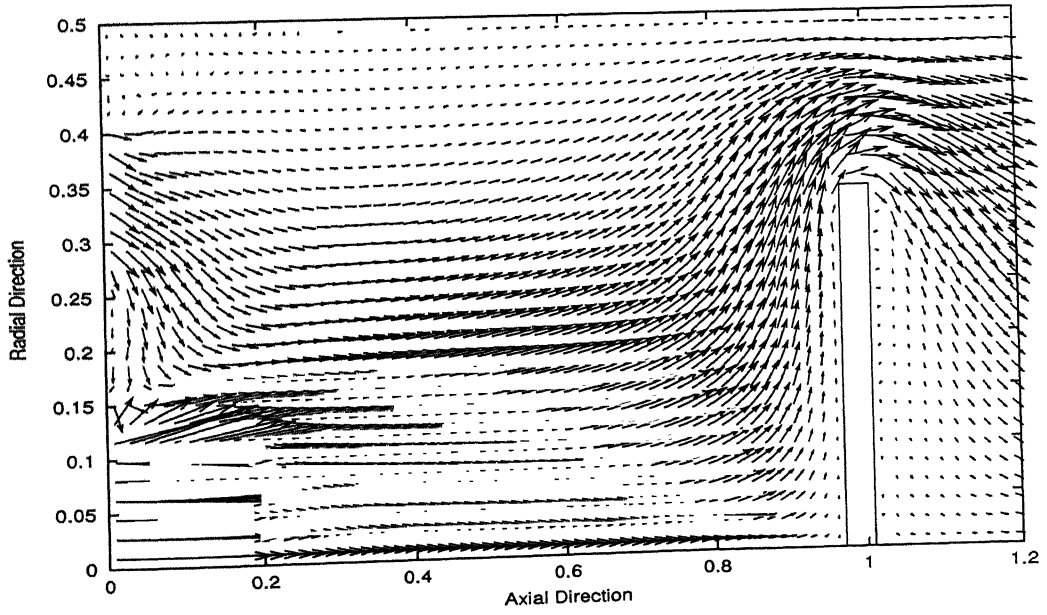


Figure 4.17: Sequentially computed vector plot on fine grid.

Skin Friction Coefficient Comparison in 3-D Straight Block, Re = 50.

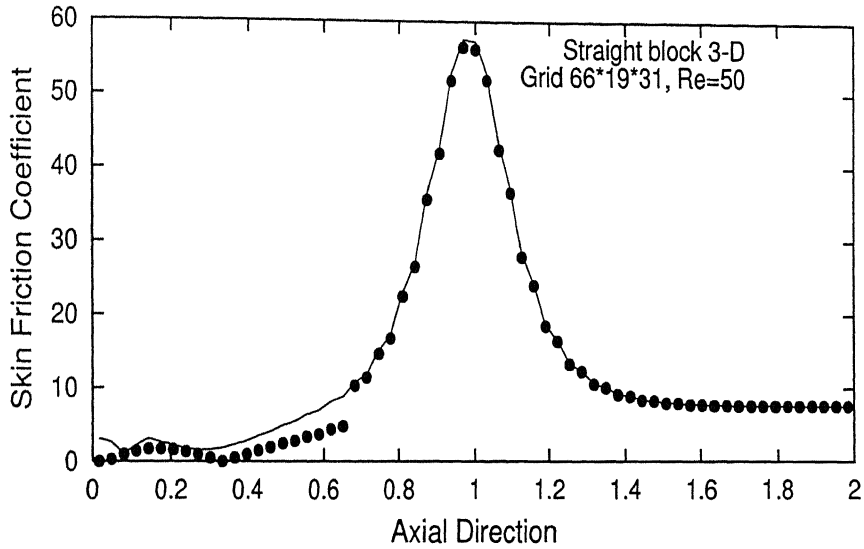


Figure 4.18: Comparison of parallely (3 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

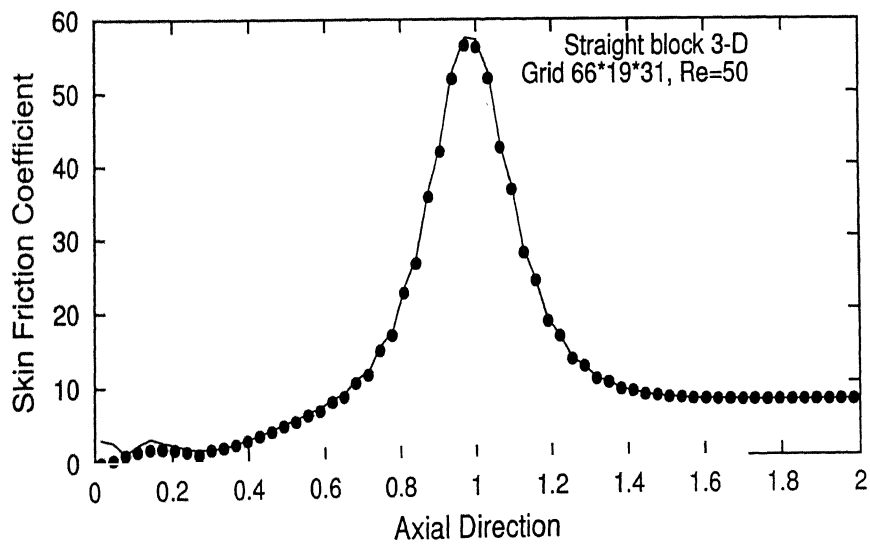


Figure 4.19: Comparison of parallely (7 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

Velocity Vector Plot for 3-D Straight Block, Re = 50.

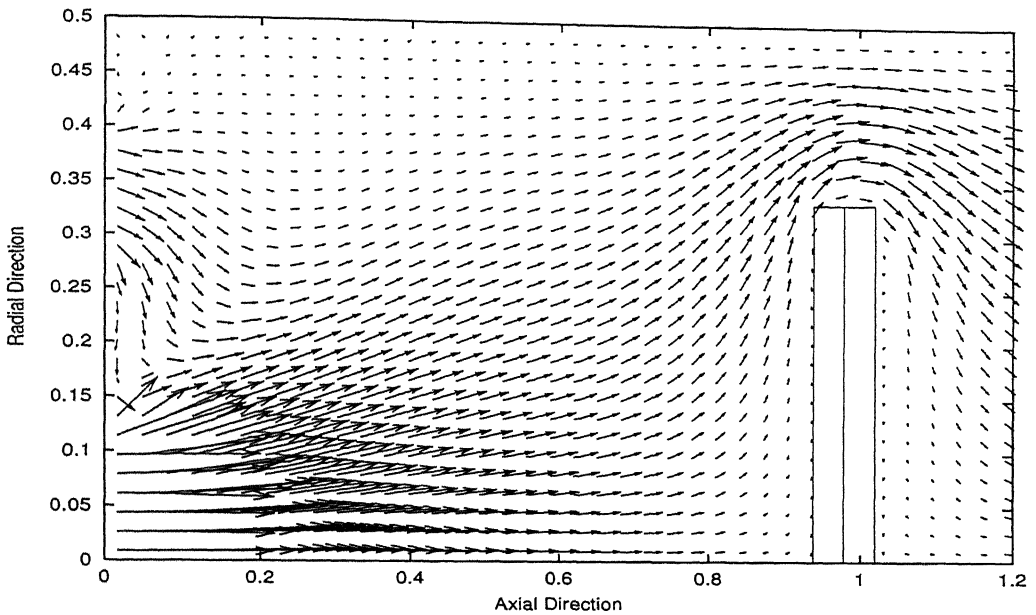


Figure 4.20: Parallely computed (3 node) vector plot in 3-D, Re 50.

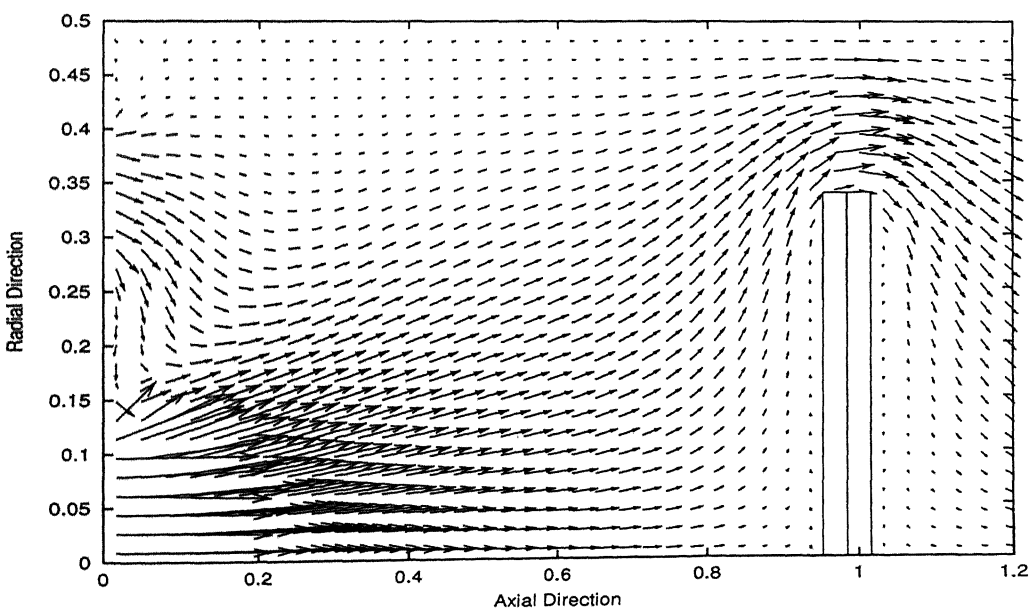


Figure 4.21: Sequentially computed vector plot in 3-D, Re 50.

Skin Friction Coefficient Comparison in 3-D Straight Block, Re = 100.

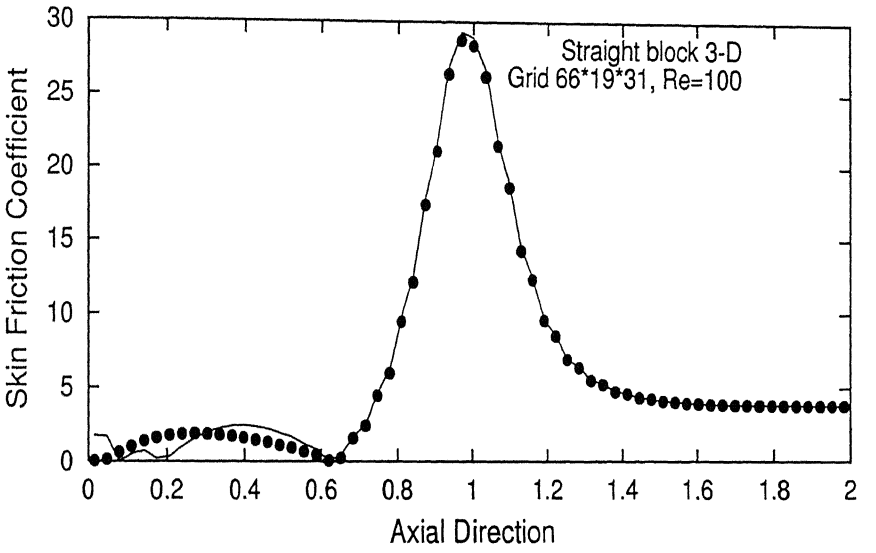


Figure 4.22: Comparison of parallely (3 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

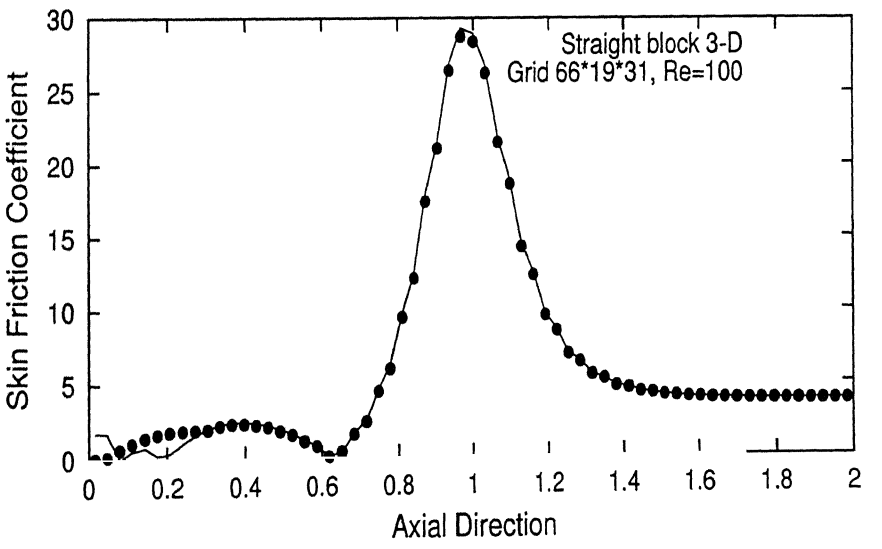


Figure 4.23: Comparison of parallely (7 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

Velocity Vector Plot for 3-D Straight Block, Re = 100.

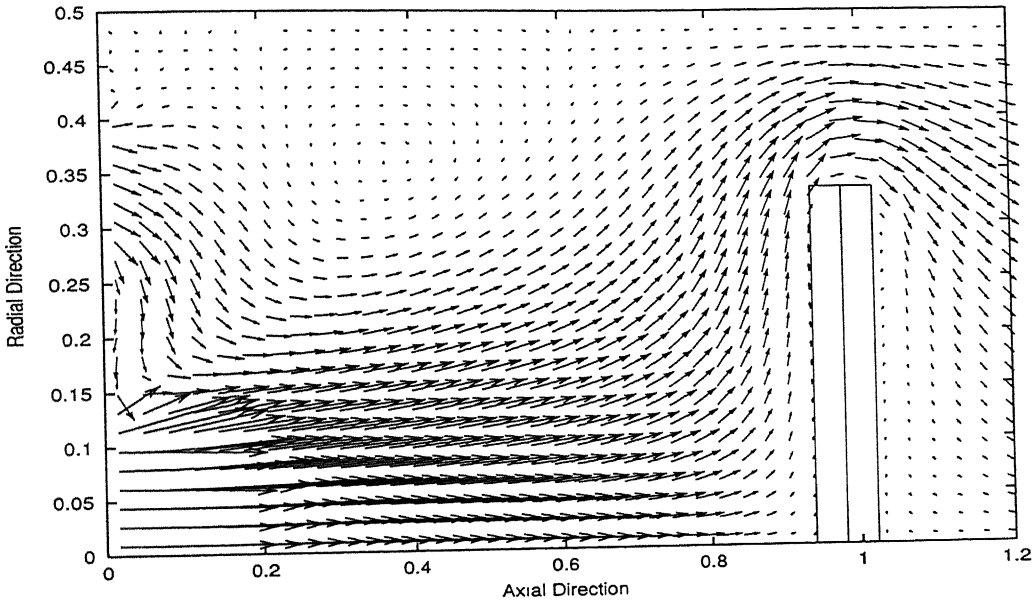


Figure 4.24: Parallely computed (3 node) vector plot in 3-D, Re 100.

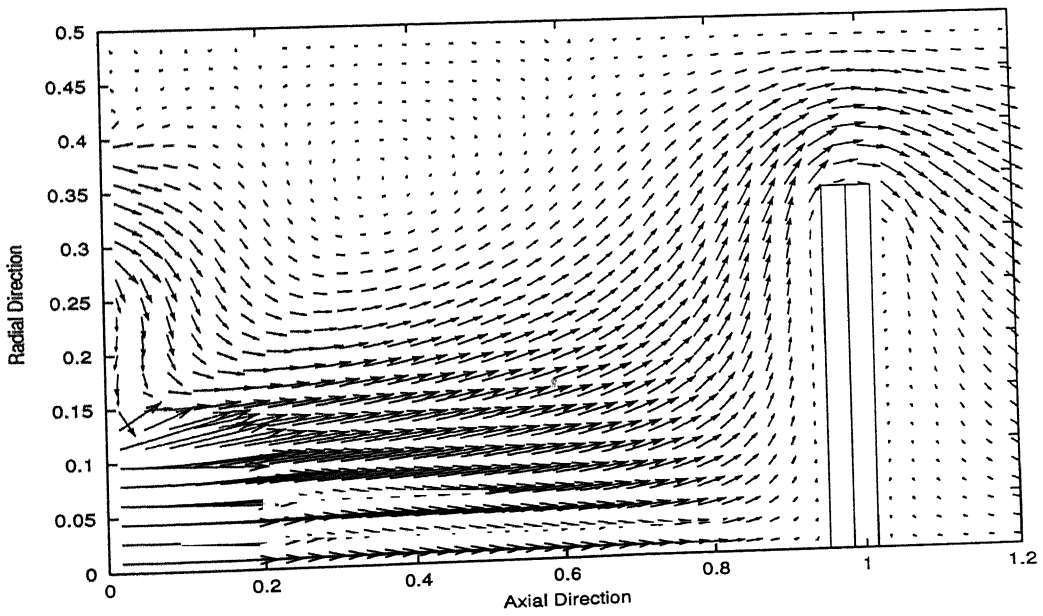


Figure 4.25: Sequentially computed vector plot in 3-D, Re 100

Skin Friction Coefficient Comparison in 2-D Curved Block, Re = 50.

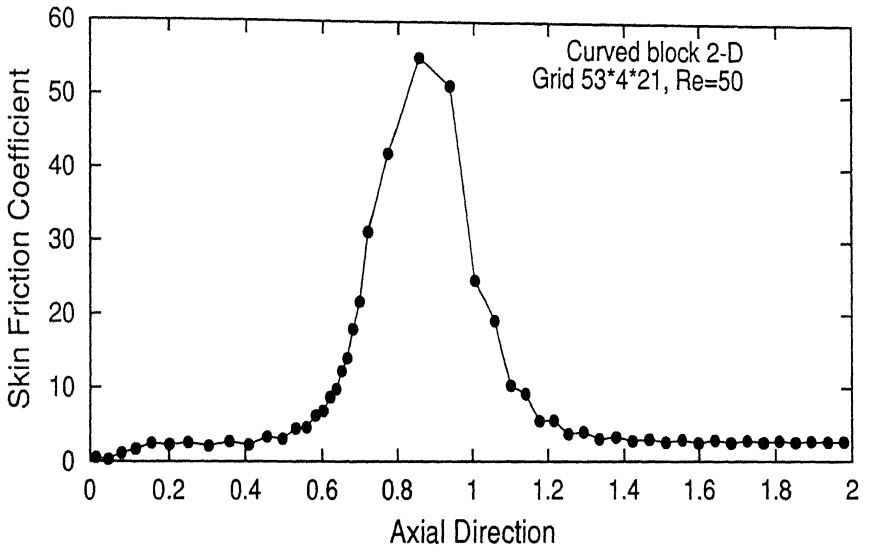


Figure 4.26: Comparison of parallely (5 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

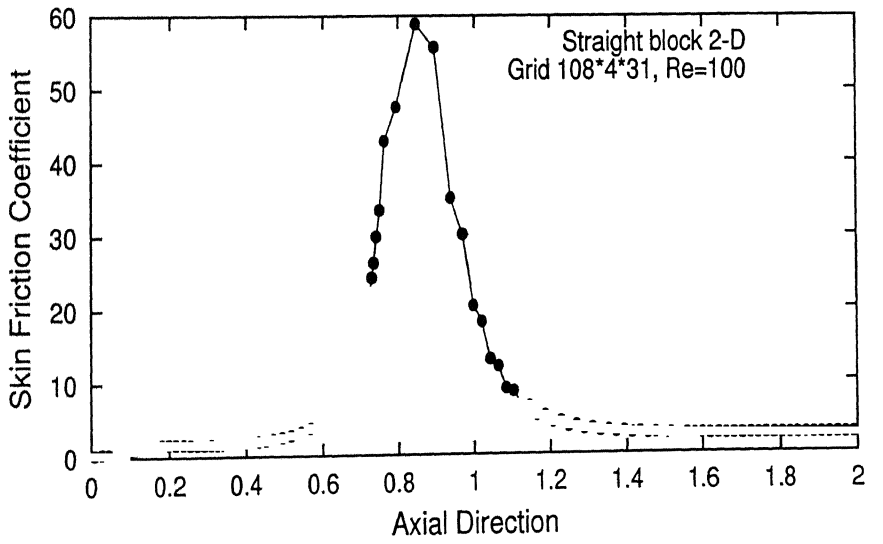


Figure 4.27: Comparison of parallely (3 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

Velocity Vector Plot for 2-D Curved Block, Re = 50.

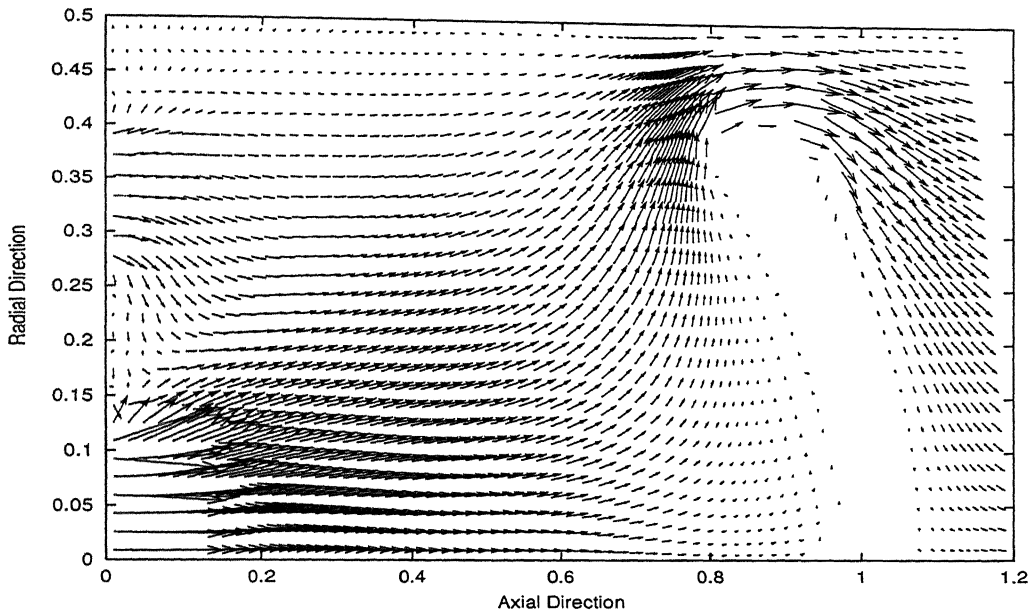


Figure 4.28: Parallelly computed (5 node) vector plot on fine grid.

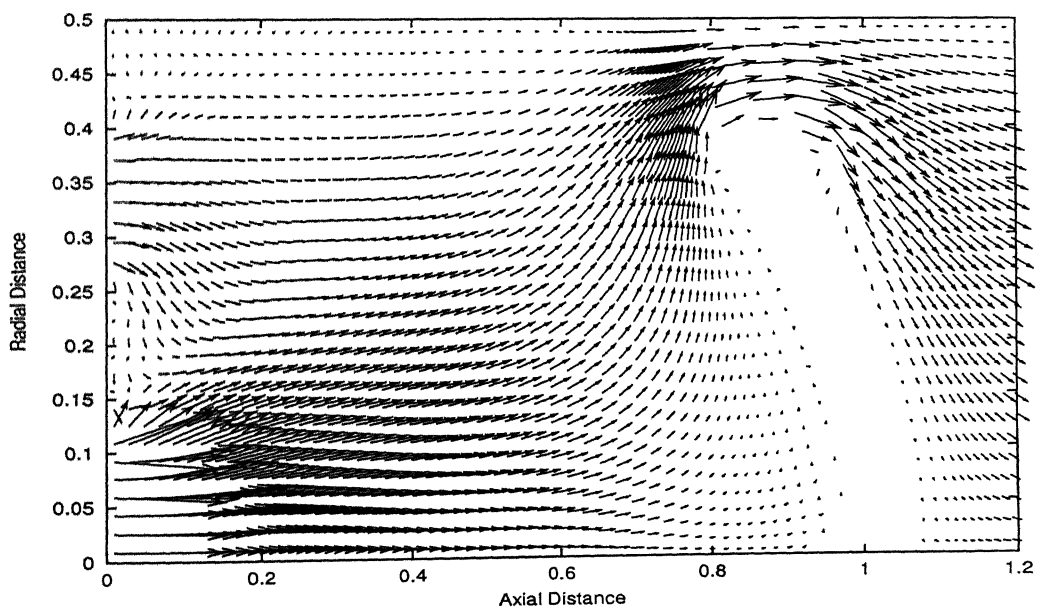


Figure 4.29: Sequentially computed vector plot on fine grid.

Skin Friction Coeffficient Comparison in 2-D Curved Block, Re = 100.

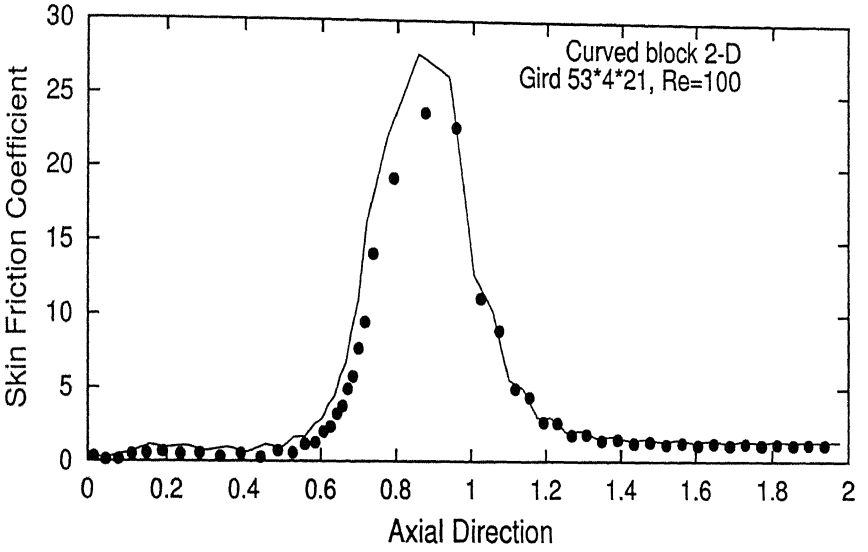


Figure 4.30: Comparison of parallely (5 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

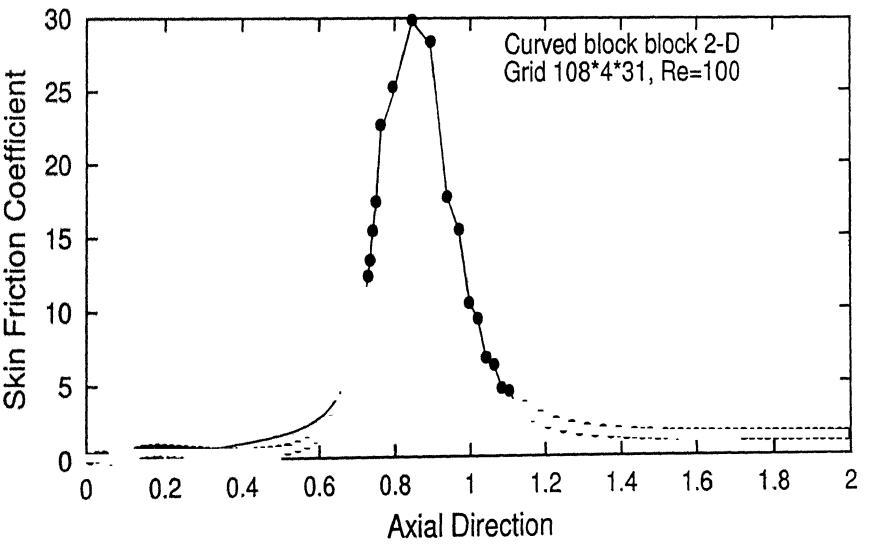


Figure 4.31: Comparison of (3 node) computed (dotted) skin friction coefficient with result obtained from 1 processor (full curve).

Velocity Vector Plot for 2-D Curved Block, Re = 100.

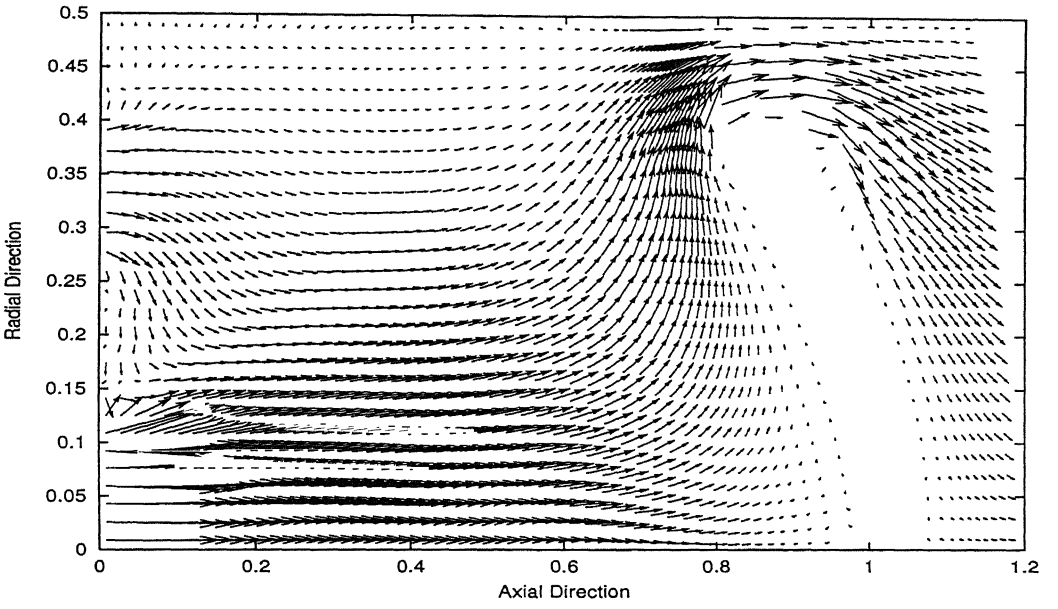


Figure 4.32: Parallely computed (5 node) vector plot on fine grid.

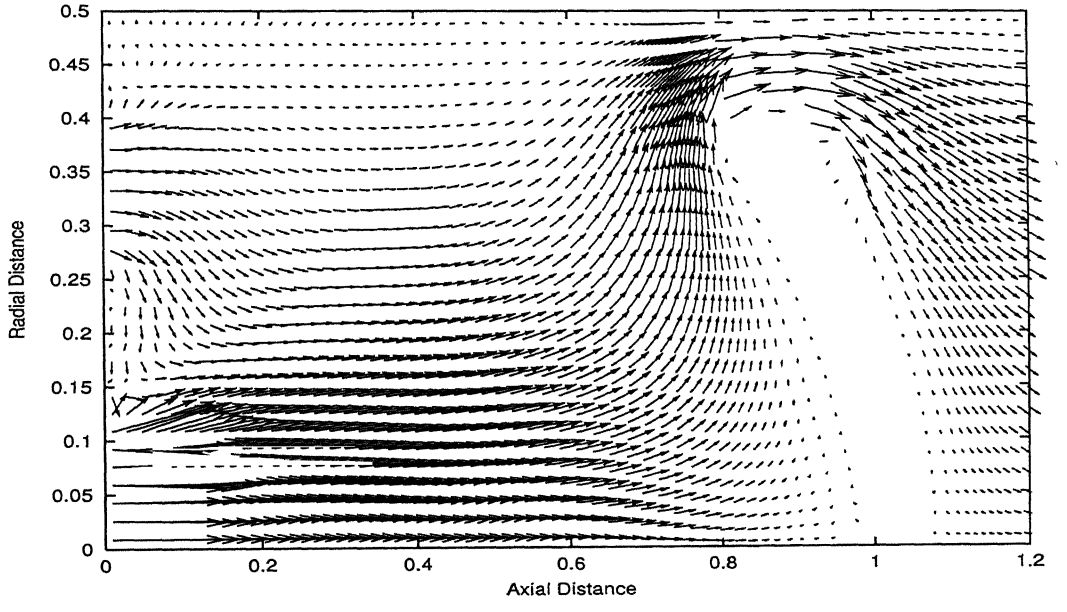


Figure 4.33: Sequentially computed vector plot on fine grid.

Chapter 5

Conclusions

At the end of this work the following conclusions can be made.

1. A Navier-Stokes solver for complex geometries was sucessfully parallelised and run on a parallel PC network using ANULIB as message passing software.

2. Parallel processing is seen to be a practical tool for handling large time consuming codes.

5.1 Scope of Future Work

1. We have shown asynchronous communication to be better for reduction in communication time and thus increasing speed-up, only on Poisson equation. The CVD code is yet to be tested on asynchronous mode.

2. In this work data parallellism inherent to the code was exploited. Any conventional domain decomposition techniques is yet to applied on this code. Performance of domain decomposition techniques compared to this data parallel model should be explored.

3. Like the momentum, both energy and mass transfer equations are basically advection-diffusion equations. So they can be made parallel based on the same model.

4. Communication calls are independent of geometry. In the present work only one geometry was tested. Problem involving other geometries must be tried.

5. In this work arrays were defined in full both in master and slaves. When we use very large size arrays it essentially diminishes the performance. To handle larger problems in parallel it is essential to define arrays only for that subdomain size plus number of the communication faces. It may be tried as part of a future work.

Bibliography

[1] ANULIB, Software Manual.

[2] Eswaran, V., Senthan, S., Biswas, G., Muralidhar, K. and Dhande S.G., *Numerical Simulation of Unsteady Three-Dimensional Flow Around an Elongated Body Moving in an Incompressible Fluid.* Turbulence Modelling, Technical Report No.4, DRDL Report, Hyderabad, July 1998.

[3] Kordulla, W. and Vinokur. M, Efficient computation of volume in flow predictions, *AIAA Journal*, Vol 21, pp 917-918 (1983).

[4] Gropp, W.D., Lusk, E., Skjellum, A.,*Using MPI - Portable Parallel Programming with the Message Passing System.* The MIT Press, Cambridge, Massachusetts, U.S.A.(1995).

[5] Gropp, W.D., Smith, E.B., Computational fluid dynamics on parallel processors, *Computers and Fluids*, Vol. 18, pp 289-304 (1990).

[6] Muralidhar, K. and Sundararajan, T. (Ed) (1995), *Computational Fluid Flow and Heat Transfer*, Narosa Publishing House, India.

[7] Ortega, J.M., Golub, G., *Scientific Computing An Introduction with Parallel Computing.* The Academic Press (1993).

[8] Quinn, J.M., *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill(1988).

[9] Topping, B.H.V., Khan, A.I. and Sziveri, J., Parallel and distributed processing for computational mechamics: an introduction. In: *Parallel and Distributed Processing For Computational Mechanics: Systems and Tools* (Edited by Topping, B.H.V.). Saxe-Coburg Publications, Edinburgh (1999).